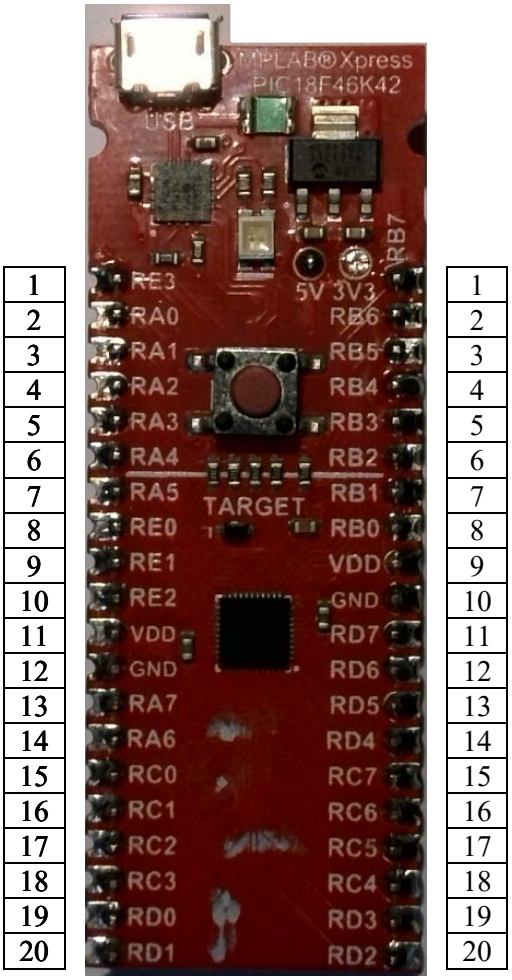
# Physics 1600

# Lecture Notes




| | |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
| 5 | 5 |
| 6 | 6 |
| 7 | 7 |
| 8 | 8 |
| 9 | 9 |
| 10 | 10 |
| 11 | 11 |
| 12 | 12 |
| 13 | 13 |
| 14 | 14 |
| 15 | 15 |
| 16 | 16 |
| 17 | 17 |
| 18 | 18 |
| 19 | 19 |
| 20 | 20 |

**Revised Summer 2020**

# Table of Contents


**INTRODUCTION** 6

**CHAPTER 1. PIC MCU** 7
- Configuring the MCU Peripherals 7
- Breadboard 9
- Wiring Your PIC/MCU and Breadboard 10

**CHAPTER 2. C PROGRAMMING** 12
- The Simplest Project 13
- Simple Math 13
- Using Math Functions 15
- Loops 16
- Comparisons and Logical Operations 18
- Arrays 20
- Functions 21
- Libraries 22

**CHAPTER 3. DEBUGGING** 25
- Breakpoints, Watch, and StopWatch 25
- Error and Warning Messages 26

**CHAPTER 4. SERIAL COMMUNICATION** 27
- Strings as Input 29

**CHAPTER 5. TEXT & STRINGS** 31
- Strings 31
- Strings as Input 32

**CHAPTER 6. NUMBERS** 35
- Overflow 36
- Storing Negative Numbers 36
- Printing Numbers in Different Formats 37
- Decimal or Floating-Point Numbers 39
- Arithmetic and Memory 40
- Text and Memory - ASCII 41
- Bitwise Operators 42
- Bit Masks 43
- Structures and Bitfields 44
- Random Numbers 46

**CHAPTER 7. REVIEW OF ELECTRIC CIRCUITS 48**
Current 48
Battery and Voltage 48
Resistors 49
Colour Code 49
Series/Parallel Connections 50
Shorting 51
BreadBoard 51
Electronics Style Circuit Diagrams 51
Kirchhoff's Rules 52
Voltage Divider 54
Potentiometer 56
Diodes 56
Examples – KR and diode circuits 57
Capacitors 59
Transducers 61

**CHAPTER 8. DIGITAL INPUT OUTPUT (DIO) 62**
Digital Output and Controls 65
Digital Input and Sensors 66
Buttons and DIO 67

**CHAPTER 9. PROCESSOR SPEED & DELAYS 74**
Changing the MCU Clock Frequency 76
Determining the Clock Frequency of the MCU 77
Using am External Oscillator 77
Delays 79
DIO and Square Wave Output 79
Timing Diagrams 80

**CHAPTER 10. TIMERS 82**
Creating Non-blocking Delays 86
Non-blocking Delays of More than One Duration 88
Timing Code Execution 88
Timing Events 91
Timers as Counters 92

**CHAPTER 11. PULSE-WIDTH MODULATION (PWM) 95**
PWM and LEDs, Motors, and Speakers 97
PWM Functions 99

**CHAPTER 12. CAPTURE 102**

**CHAPTER 13. SIGNAL MEASUREMENT TIMER (SMT) 108**
Timer Functions 108
SMT Counter 112
SMT Capture 114
SMT Specialty Capture Modes 118

**CHAPTER 14. ANALOG-TO-DIGITAL CONVERSION (ADC) 128**
ADC on Time Dependent Input Signals 130
Configuring ADC with MCC 131
Using External References 134
Precision Reference 135
ADC and Transducers 136

**CHAPTER 15. ANALOG OUTPUT (DAC) 138**

**APPENDIX A. SOFTWARE 142**
Installing the Programs at Home 142
Plug-ins 144
Creating a Project 145
Timesaver 152
C90 or C99 153

**APPENDIX B. THE SIMULATOR 154**
Breakpoints 156
Watching Variables 157
Stopwatch 159

**APPENDIX C. PUTTY 160**
Configuring PuTTY on the PC 160
Controlling the PuTTY Terminal Programmatically 163
Simple Serial Terminal 163
Data Visualizer 164

**APPENDIX D. ASCII 168**
ASCII Control Characters 169

**APPENDIX E. THE OSCILLOSCOPE 171**
Using the Oscilloscope 174
Reference Voltage and Frequency 174
USB Flash Drive 174

**APPENDIX F. GRAPHING WITH MS EXCEL 175**

**APPENDIX G. SPARKFUN LCD 178**
Connecting the serLCD 178
Special Commands 178

**APPENDIX H. XC8 AND MCC LIBRARIES 181**
**math.h 181**
**stdio.h 181**
**ctype.h 182**
**stlib.h 182**
**strings.h 183**
**Peripheral Functions and SFRs from MCC 184**
**Delays 184**
**UART 184**
**DIO 184**
**Timers 184**
**PWM 185**
**Capture 186**
**SMT 186**
**ADC 187**
**DAC 187**
**FVR 188**

# Introduction

The Integrated Circuit (IC) has become pervasive in modern society. Once associated only with Electronics Engineering, the modern IC has evolved to the point where it has become a standard tool in all areas of science and engineering. Every practical physicist engineer needs an understanding of the capabilities of these devices. In this course you will learn the fundamentals of how a Programmable Integrated Circuit/Microcontroller Unit (PIC/MCU) works. Working with a high-level programming language, C, you will make a MCU handle a wide variety of tasks including reacting to external stimuli. Along the way you will learn basic electronics, interfacing with sensors, and oral presentation skills.

# Chapter 1. PIC MCU

The microcontroller unit (MCU) is simply a computer designed to work with other machines rather than people. Because of this, they do not need a monitor, keyboard, mouse, or any other device that requires human interaction.

Most MCUs have a CPU, RAM, ROM, and some EEPROM (Electrically Erasable Programmable Read Only Memory). Inexpensive controllers that are designed to be programmed once (for a particular application) may omit the EEPROM, because there may be no need to ever change the program once the MCU goes "into production".

The MCU you will use, the PIC18F46K42, has 64kb of flash memory which allows you to erase and reprogram it about at least one hundred thousand times, so you do not need to worry about "wearing out" the chip. This flash memory can hold your program and any data storage required in your application. Technical details are available in the datasheet.

You will create the circuits you will use with the MCU on a "breadboard". On a PC, using a program called the MPLAB X IDE, you will create and test the programs that will run on the MCU. The programs will be written using the C language. When finished, you will transfer the program to the MCU using a USB connection and a device called a programmer. The programmer is a second MCU. The PIC18F46K42, the programmer, and a USB connector are all located on an MPLAB Xpress board for convenience. One could purchase the PIC18F46K42 and the programmer separately.

The PIC18F46K42 is a 40-pin device. Some of the pins are used to connect a battery or power supply to the MCU. The rest of the pins are for inputs and outputs. The pin-out diagram of the PIC18F46K42 is shown in Figure 1-1 below. Note the notch on the chip. This orients the chip and indicates the top of the chip and which pin is number 1. The PIC18F46K42 has many features, called peripherals, built into the chip including digital input and output (DIO), timing and square wave production, serial communication, and analog-to-digital conversion (ADC) that you will learn to use. Depending on the feature of the MCU being used, the pins have different functions and names (this is called *multiplexing*).

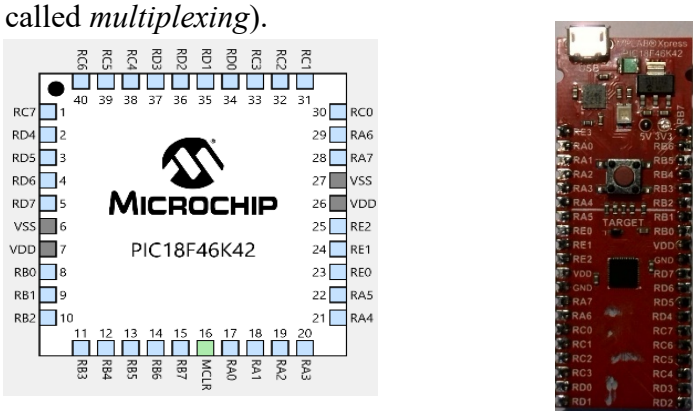



**Figure 1-1:** Pin out schematic of the PIC18F46K42 and the Xpress Board.

## Configuring the MCU Peripherals

Selection and configuration of a peripheral involves *Special Function Registers* or SFRs. These are special portions, eight bits wide, of MCU memory. Some bits act as ON/OFF or TRUE/FALSE switches that you set (1/ON/TRUE) or clear (0/OFF/FALSE) for different aspects of the operation of the peripheral. Some bits are not used at all. Some bits you can only read and not change. Some SFRs will hold the results of a measurement. Typically, a peripheral will involve using a number of SFRs. For example, the UART2 peripheral we use for serial communication has nineteen SFRs. Each SFR for each peripheral is itemized in the PIC18F46K42 data sheet. An example from the datasheet, for the UART Control Register 0, UxCON0, is show in the figure below. The x stands for either 1 or 2 since there are two UART peripherals for this PIC. Each bit

| Peripheral | Bit Name Prefix |
|---|---|
| UART 1 | U1 |
| UART 2 | U2 |

**REGISTER 31-1: UxCON0: UART CONTROL REGISTER 0**

| R/W-0/0 | R/W/HS/HC-0/0 | R/W-0/0 | R/W-0/0 | R/W-0/0 | R/W-0/0 | R/W-0/0 | R/W-0/0 |
|---|---|---|---|---|---|---|---|
| BRGS | ABDEN | TXEN | RXEN | MODE[3:0] | | | |
| bit 7 | | | | | | | bit 0 |

| Legend: | | |
|---|---|---|
| R = Readable bit | W = Writable bit | U = Unimplemented bit, read as '0' |
| u = Bit is unchanged | x = Bit is unknown | -n/n = Value at POR and BOR/Value at all other Resets |
| '1' = Bit is set | '0' = Bit is cleared | HC = Hardware clear |

bit 7 **BRGS:** Baud rate Generator Speed Select bit
1 = Baud rate generator is high speed with 4 baud clocks per bit
0 = Baud rate generator is normal speed with 16 baud clocks per bit

bit 6 **ABDEN:** Auto-baud Detect Enable bit(3)
1 = Auto-baud is enabled. Receiver is waiting for Sync character (0x55)
0 = Auto-baud is not enabled or auto-baud is complete

bit 5 **TXEN:** Transmit Enable Control bit(2)
1 = Transmit is enabled. TX output pin drive is forced on when transmission is active, and controlled by PORT TRIS control when transmission is idle.
0 = Transmit is disabled. TX output pin drive is controlled by PORT TRIS control

bit 4 **RXEN:** Receive Enable Control bit(2)
1 = Receiver is enabled
0 = Receiver is disabled

bit 3-0 **MODE[3:0]:** UART Mode Select bits(1)
1111 = Reserved
1110 = Reserved
1101 = Reserved
1100 = LIN Master/Slave mode(4)
1011 = LIN Slave-Only mode(4)
1010 = DMX mode(4)
1001 = DALI Control Gear mode(4)
1000 = DALI Control Device mode(4)
0111 = Reserved
0110 = Reserved
0101 = Reserved
0100 = Asynchronous 9-bit UART Address mode. 9th bit: 1 = address, 0 = data
0011 = Asynchronous 8-bit UART mode with 9th bit even parity
0010 = Asynchronous 8-bit UART mode with 9th bit odd parity
0001 = Asynchronous 7-bit UART mode
0000 = Asynchronous 8-bit UART mode

**Note 1:** Changing the UART MODE while ON = 1 may cause unexpected results.
**2:** Clearing TXEN or RXEN will not clear the corresponding buffers. Use TXBE or RXBE to clear the buffers.
**3:** When MODE = 100x, then ABDEN bit is ignored.
**4:** UART1 only.

**Figure 1-2: Typical SFR information**

You can program the SFR to have your choice of feature using assigning a numerical value to the entire SFR as shown below

```
#include <xc.h>    //C library for SFRs

U2CON0 = 0b10100110;  // your choices – binary makes clears
                      // which bit has which value
```

Single bits and groups of bits can be addressed directly by the secondary names shown in figure above. The previous code could be rewritten

```
#include <xc.h>    //C library for SFRs

U2CON0bits.BRGS = 1;
U2CON0bits.ABDEN = 0;
U2CON0bits.TXEN = 1;
U2CON0bits.RXEN = 0;
U2CON0bits.MODE = 0b0110;
```

The general format is SFRNAMEbits.BITSNAME. Generally, we will not work at the SFR level. We do not need that level of control and most of our choices for the peripheral are quite simple. Also figuring out all the settings of all the SFRs for a particular peripheral is quite daunting to a new user of an MCU. Fortunately, Microchip provides an alternate, and much simpler way, of configuring the SFR values for a peripheral with a graphical interface in the MPLAB X IDE call the MPLAB Code Configurator or MCC.

## Breadboard

A breadboard is a convenient way to assemble circuits where the parts are small such as microchips, diodes, and resistors. Pinholes in the front of the breadboard let you securely connect circuit elements to it. The sets of pinholes are connected by conducting bars in two different patterns, short horizontal and long vertical, under the plastic cover. See Figure 1-3. This setup allows parallel and series arrangements of elements to be put together quickly. When using wire connectors make sure to keep them short and neat!

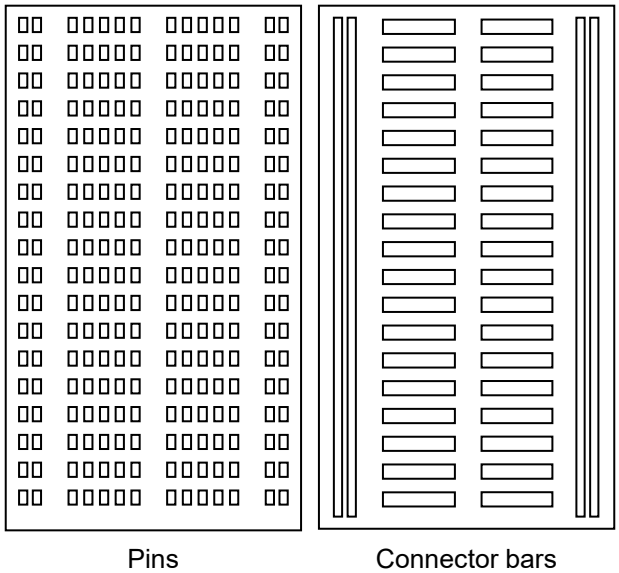



**Figure 1-3:** Breadboard connectors.

We will start this course with an introduction to C programming and the particulars of programming an MCU (microcontroller) unit. We will lay a lot of emphasis on understanding how numbers (integers) are stored in memory.

We will also review basic electrics circuit and Kirchhoff's Rules. We will also cover the simplified notation used in electronics for circuit diagrams.

Separately we will learn how to use special modules on the MCU including timing, DIO, ADC, PWM, and Capture.

## Wiring Your PIC/MCU and Breadboard

Your PIC/MCU is on a red circuit board with additional electronics to facilitate getting power to the PIC/MCU from a USB connection to your PC, programming your PIC/MCU from your PC, and serial communication between the PIC/MCU and your PC. This is the MPLAB Xpress package. The Xpress board snugly fits onto the top centre of a breadboard (Figure 1-4).

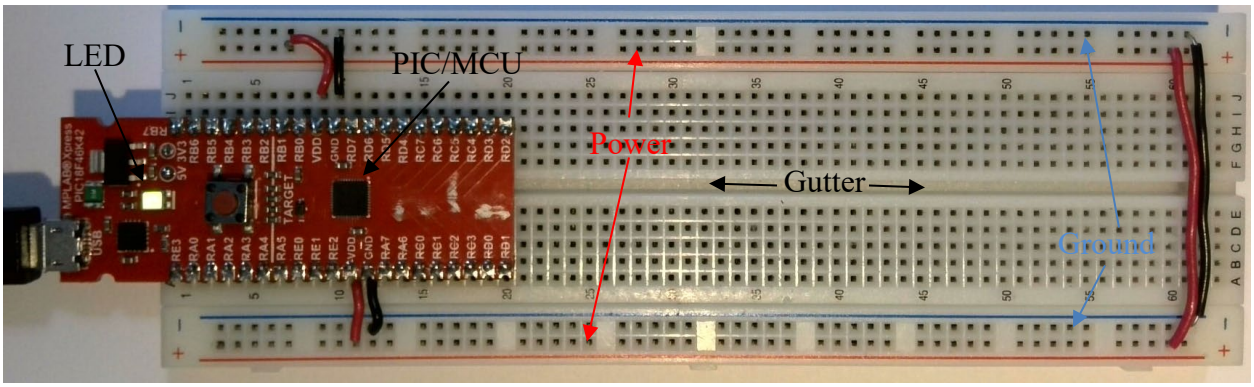



**Figure 1-4:** PIC/MCU in MPLAB Xpress Board on breadboard.

The breadboard is where you make connections. Each hole is a little clamp; inserting the bare end of a wire into a hole makes a secure connection. There are two types of groups or sets of holes. On the sides, indicated by red and blue lines, are the power and ground lines. The red side is "hot" or "high" at +5.0 Volts. The blue side is "ground" or "low" at 0 Volts. Electric circuits are conducting (i.e. wired) paths from high to low passing through circuit elements (resistor, LEDS, and PICs, etc.).

Your PIC/MCU straddles a gutter in the breadboard. Each leg or pin of the PIC/MCU is in a different 5-hole connecter. Your PIC/MCU is powered by the USB connection. You power the breadboard by connecting the $V_{DD}$ pins on either side to the power line on each side. Similarly, the $V_{SS}$ pins (one on either side) must be connected to a ground line on either side. Use red wire for power and black for ground. Consult the pin out diagram and make sure you know where these pins are. When you finish your breadboard can now be used to build working circuits.

The numbered holes in the centre are where connections are made. If you want to connect two wires together, you connect them to different slots of the same group of five holes. The diagram below, Figure 1-5, shows two different ways of connecting a resistor, and an LED, and another resistor in series between power and ground so that the LED lights up.

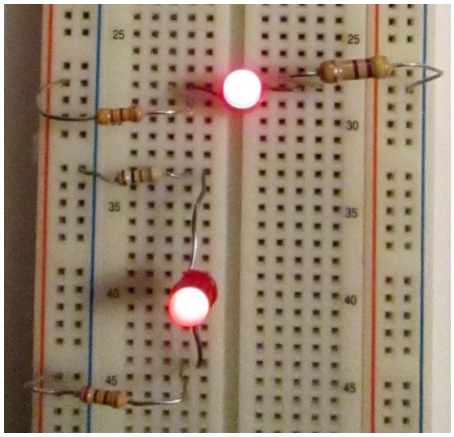



**Figure 1-5**: Series circuits

The board has not had its wiring finished. The $V_{DD}$ and $V_{SS}$ pins must be connected to the power and ground rails. Part of the reason for this is to provide you with practice stripping wires and wiring the breadboard. We will start with our (very cheap) yellow wire strippers and practice removing about 1 cm of plastic coating from the ends of a piece of wire. We do not want any nicks in the wire; that weakens the wire and makes it prone to breaking! You can also cut the wire with the strippers.

# Chapter 2. C Programming

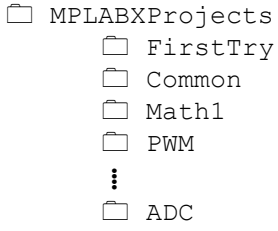
You may already have taken a programming course such as C, C++, Visual Basic or Java. That certainly helps in this course, but it is not a requirement. We will teach you all the programming you need to make a PIC MCU perform all the tasks we will give you. We will be using C to program the PIC MCU. You will need to learn the statements and syntax of commands and functions but the level we will take you is low. On the Moodle site we have links to C programming eBooks. You will only need to read and master the topics in only about the first four or five chapters of these books which have twenty or more chapters. You can stick to whichever eBook you like best.

The version of C you will be using, XC8, is specifically designed for use with the Microchip PIC18 chips. As such commands to use the special features or your chip such as PWM, ADC, and Timers are not covered in those books. Instead you will find notes and example code in dedicated chapters in this text.

You will be creating projects. A project is a collection of one or more files that you will write, or that we supply, or that are part of the XC8 package. For the most part these files are just plain text files, i.e. you can read them, but with the ending .c or .h instead of .txt. Typically in C, the compiler builds a program or .exe file from your project code. In XC8 the compiler turns the project into a .hex file that the PIC MCU can understand. For the most part you will be writing your files in the Microchip MPLAB X Integrated Development Environment, IDE for short. The IDE is a wonderful tool for writing your files, alerting you to spelling mistakes, and missing information.

It is easy to get overwhelmed by all the settings and vocabulary you need to learn at the start. Don't panic! Persevere and try to work in groups.

We expect everyone to use a USB memory stick or perhaps Cloud memory. Keep your files in a sensible order. We recommend the following structure and use folder names that make it easier to find your projects:



## The Simplest Project

The simplest project consists of a single .c or source code text. The code looks like this

```
int main(void)
{
  return 0;
}
```

**Figure 2-1:** A basic program.

Every project, containing one file or many, has a starting point called the main() function. You can only have one main().

If you have installed the MPLAB X IDE and XC8 Compiler, you can follow the steps in *Appendix AAppendix A Software* and *Appendix C The Simulator* to build a project containing this file and run it in Debugger|MPLAB Sim mode.

## Simple Math

Physicist use computers primarily to do calculations. XC8 knows the basic operations addition, subtraction, multiplication, division, and modulus, the symbols of which are +, -, *, /, and %. If you wanted to write an expression to find the sum of two numbers, you write it just as you would normally, but you must end with a semicolon as below.

```
int main(void)
{
 // these slash marks allow you to add comments
 a = 5 + 2;  // note ; at end of line!!
 b = 5 - 2;
 c = 5 * 2;
 d = 5/2;
 e = 5%2;    // remainder of 5 divided by 2
 return 0;
}
```

**Figure 2-2:** Basic math operations

Although all the lines we wrote are correct C, this project will crash. C needs to put numbers, variables, and functions into memory. It needs to know how much memory to assign. We have not told the compiler how much room to leave for the variables a, b, c, d, and e. Variables holding number variables can be one of three types of integer called `char`, `int`, and `long` and one decimal type called `float` or `double`. Type `char` can hold 8 bits, each bit being a 0 or 1, and can hold number from 0 to 255. Type `int` has 16 bits, room for numbers from 0 to 65535. Type `long` has 32 bits and can hold numbers in the range of 0 to 4,294,967,295. Why do we need so many types of integer? PIC MCUs like early computer have limited memory space. Suppose you can have only 400 variables of type `char` before you run out of space, then you could only have 200 variables of type `int`, or 100 variables of type `long`. Variable of type float require 32 bits of memory and range from -2.77000e+37 to +2.77000e+37 with about seven significant digits.

Since the actual numbers we use are integers, let us use integers of type `int`.

```
int main(void)
{
 int a, b, c, d, e = 10;  // you can initialize a variable
 a = 5 + 2;  // note ; at end of line!!
 b = 5 - 2;
 c = 5 * 2;
 d = 5/2;
 e = 5%2;
 return 0;
}
```

**Figure 2-3:** Variables must be assigned memory.

Even though this project compiles it is unexciting; we do not see the results of the calculations.

In standard C, we could use the `printf()` function from the C library `stdio.h` to print to a display. So the following would work.

```
#include <stdio.h> // library that has printf()

int main(void)
{
 int a, b, c, d, e = 10;  // you can initialize a variable
 a = 5 + 2;  // note ; at end of line!!
 b = 5 - 2;
 c = 5 * 2;
 d = 5/2;
 e = 5%2;

 printf("a = %u, b = %u, c = %u, d = %u, e = %u \n\r",a,b,c,d,e);
 return 0;
}
```

**Figure 2-4:** How to print in C but not XC8

In `printf()`, the part between the quotes is called a format string. The %u symbols are placeholders that tell the compiler to convert the corresponding variable to an integer number string.

While this code compiles correctly, the Simulator still does not show anything. The reason for this is that the Simulator simulates the PIC MCU and it does not have a display. Instead the PIC MCU has a UART module that can be configured to send text to a display if one is wired to the PIC MCU correctly. The Simulator will play the part of the appropriately wired display as long as the code properly configures the UART module. This is illustrated in *Chapter 4 Serial Communication*. The MPLAB Code Configurator (MCC) creates the necessary code set the operatings speed of the PIC/MCU and the parameters of the UART. Everything needed is in a function SYSTEM_Initialize(). Ignoring the details for now, the code that does produce `printf()` output is

```
#include "mcc_generated_files/mcc.h"  // library for SYSTEM_Initialize
#include <stdio.h> // library that has printf()
```

```
int main(void)
{
 // Initialize the device - MCC file configures clock and UART
 SYSTEM_Initialize();

 int a, b, c, d, e = 10;   // you can initialize a variable
 configureUART(9600ul,1); // opens serial communication
 a = 5 + 2;  // note ; at end of line!!
 b = 5 - 2;
 c = 5 * 2;
 d = 5/2;
 e = 5%2;

 printf("a = %u, b = %u, c = %u, d = %u, e = %u \n\r",a,b,c,d,e);
 return 0;
}
```

**Figure 2-5:** Printing in XC8

Besides writing equations is an algebraic style we are used to from school, C also has a number of shortcuts that we can use as shown below:

```
#include "mcc_generated_files/mcc.h"  // library for SYSTEM_Initialize
#include <stdio.h> // library that has printf()

int main(void)
{
 // Initialize the device - MCC file configures clock and UART
 SYSTEM_Initialize();

 int a = 10, b = 5, c = 3, d = 2;   // you can initialize a variable
 configureUART(9600ul,1); // opens serial communication

 a++;       // same as a = a + 1
 b--;       // same as b = b - 1
 c += 5;    // same as c = c + 5
 d *= 3;    // same as d = d * 3

 printf("a = %u, b = %u, c = %u, d = %u \n\r",a,b,c,d);
 return 0;
}
```

**Figure 2-6:** Math shortcuts

## Using Math Functions

Besides the simple math operations illustrated above, you also can use all the math functions that you are use on your calculator such as sine, square root, and exponential, etc. These functions are kept in the math.h library so you need to include that file in you code when you plan to make use of those functions. A complete list of the functions available can be found in *Appendix H XC8* Libraries.

```
#include "mcc_generated_files/mcc.h"  // library for SYSTEM_Initialize
#include <stdio.h> // library that has printf()
#include <math.h>  // library for common math functions
int main(void)
{
 // Initialize the device – MCC file configures clock and UART
 SYSTEM_Initialize();

 float a = 10, b = 5, c = 3, d = 2;   // you can initialize a variable
 configureUART(9600ul,1); // opens serial communication

 a = 5.0*sin(0.1);
 b = sqrt(14.2);
 c = pow(5,3);       // same as c = (5)**3
 d = log(3.2);       // log means natural log ln

 printf("a = %f, b = %f, c = %f, d = %f \n\r",a,b,c,d);
 return 0;
}
```

**Figure 2-7:** Math.h functions

Note that results of these functions are decimal numbers, so the variables have been declared as `float`. Also, we use %f to print decimal numbers.

## Loops

C has three loop functions, the for loop and the while and do-while loops. The for loop is useful for counting or summing. The while and do-while loops keep looping until a condition is met; think of them as a wait until function.

The form of the for loop is

```
for( count = start; count <= end; count = count + increment)
    {
      // your code goes here
      // never alter count inside here
    }
```

The variables `count`, `start`, `end`, and `increment` are integer variables, `char`, `int` or `long` as necessary. Normally `count` is the only variable you define, `start`, end and `increment` are usually numbers. The loop continues as long as the conditions are true. It will not execute if your `end` was less than your `start`. The following example shows how to sum all the integers from zero to twenty.

```
#include "mcc_generated_files/mcc.h"  // library for SYSTEM_Initialize
#include <stdio.h> // library that has printf()

int main(void)
{
// Initialize the device – MCC file configures clock and UART
 SYSTEM_Initialize();
```

```
 int i, sum = 0;
 configureUART(9600ul,1); // opens serial communication


 for ( i=0; i <= 20; i++ )  // from 0 to 20 inclusive, increments of 1
   {
    sum = sum + i;  // could write sum += i;
   }

 printf("sum = %u \n\r",sum);
 return 0;
}
```

**Figure 2-8:** A for loop example

We will typically only use the while loop function rather than the do-while loop. And most of the time is as a wait function. The structure of the while loop is

```
while(logical condition)
   {
     // the logical condition may change externally or inside the loop
   }
```

And the structure of the do while loop is

```
do {
     // the logical condition may change externally or inside the loop
   } while(logical condition)
```

If the logical condition is true, the loop proceeds. The logical condition is either TRUE which is the same as 1 or FALSE which is 0. The difference between a while loop and a do while loop is that the while loop checks if the logical condition before executing the code in the loop and the do while loop checks after. As a result a do while loop always executes the code inside at least once.

Here is an example of a while loop using a Timer.

```
#include "mcc_generated_files/mcc.h"  // library for SYSTEM_Initialize
#include <stdio.h> // library that has printf()

int main(void)
{
  // Initialize the device – MCC file configures clock and UART
 SYSTEM_Initialize();

 TMR0_WriteTimer0(0); // starts timer at 0
 TMR0IF = 0;  // clear or zero TMR0IF
 while(!TMR0IF)  // TMR0IF changes from 0 -> 1 when timer overflows
   {
     // code to do things until Timer runs out
   }
 return 0;
}
```

**Figure 2-9:** A while loop example

The exclamation mark means NOT and NOT FALSE is TRUE and NOT TRUE is FALSE.

## Comparisons and Logical Operations

Decision making is a big part of writing useful code. The basic comparisons are less than, greater than, and equal to. C uses the familiar symbols of < for less than, > for greater than, but == for equal to. Less than or equal to is denoted by <= while greater than or equal to is >=. When a comparison is TRUE the comparison returns a 1, while if it is FALSE it returns a 0. There is also the NOT operator, !, that turns TRUE to FALSE and vice versa.

Typically, a comparison is used inside an if-else statement like below

```
if (  i == j )  // check if i and j are equal
   {
     // if true, have lines of code here
   }
else
   {
     // if false, have lines of code here
   }
```

You can make multiple comparisons with a switch statement.

```
switch(i)
   {
     case 0:    //lines of code if i == 0
                   break;
     case 1:    //lines of code if i == 1
                   break;
     case 2:    //lines of code if i == 2
                   break;
     default:   // if i is anything else
                   break;
   }
```

It is possible to make very complex comparisons, using the AND (&&) and the OR (||) operators, For example any i between 51 and 100 will satisfy the expression ( i > 50 && i <= 100) and any i greater than 4 but not odd satisfies ( i >= 5 || i%2 != 1) . This following code shows various examples of logical comparisons.

```
#include "mcc_generated_files/mcc.h"  // library for SYSTEM_Initialize
#include <stdio.h> // library that has printf()

int main(void)
{
// Initialize the device – MCC file configures clock and UART
SYSTEM_Initialize();

unsigned int i=10, j=20, k = 30;

//in C, TRUE is represented by the number 1. FALSE is usually shown as 0
//but in fact is any number other than 1
```

```
//Try to predict which statement will print

// == means is equal to
if ( i == j )  // note the double equals sign, bad errors if you drop one!
   {
    printf("TRUE i and j are equal. \n");
   }
else
   {
    printf("FALSE i and j are not equal. \n");
   }

// != means is not equal to
if ( i != j )
   {
    printf("TRUE i and j are not equal. \n");
   }
else
   {
    printf("FALSE i and j are equal. \n");
   }

//similarly but not shown are
// > means is greater than
// >= means is greater than or equal to
// < means is less than
// <= means is less than or equal to


// ! the NOT operator, makes a TRUE statement FALSE and vice versa
if ( !(i == j) )  // note the double equals sign, bad errors if you drop one!
   {
    printf("TRUE i and j are not equal. \n");
   }
else
   {
    printf("FALSE i and j are equal. \n");
   }

// Compound expressions

// && the AND operator, but statements must be true
if ( (i == j) && (j == k) )  // both must be true
   {
    printf("TRUE i and j and k are equal. \n");
   }
else
   {
    printf("FALSE either i is not equal to j or it isn't equal to k. \n");
   }

// || the OR operator, at least one statement must be true
if ( (i == j) || (j == k) )  // only one must be true
   {
    printf("TRUE but which statement is true? \n");
   }
else
   {
    printf("FALSE neither statement is true. \n");
   }
```

```
// Warning!!!
// == means is equal to, = assigns one value to another
if ( i = j )  //
   {
    printf("TRUE you assigned j to i and the value of i is 1 . \n");
   }
else
   {
    printf("FALSE you assgned j to i and the value of i was not 1. \n");
   }


// use switch to examine multiple cases
switch(i)
{
 case 0:  printf("i is 0 \n");
          break;
 case 1:
 case 2:
 case 3:
 case 4:
 case 5:  printf("i is between 1 and 5 \n"); //leave out breaks and cases
                                             //cascade into one another
          break;
 case 6:  printf("i is 6 \n");
          break;
 default: printf("i > 6 \n"); // any nonspecified case value
          break;
}


while(1)
 {
 }
 return 0;
}
```

**Figure 2-10:** Logic operations and comparisons.

## Arrays

Arrays are lists of variables. Suppose you have four test scores, you could define one separate variable for each test score or one array of length 4. Arrays can be of any type we have already seen; `char`, `int`, `long`, or `float`. In C counting starts at 0, so array[5] has five elements; array[0], array[1], array[2], array[3], and array[4].

```
#include "mcc_generated_files/mcc.h"  // library for SYSTEM_Initialize
#include <stdio.h> // library that has printf()

int main(void)
{
// Initialize the device – MCC file configures clock and UART
SYSTEM_Initialize();

int i=0, j=0;
float scores[4] = { 89.5, 90.2, 78.6, 93.0 }; // how to initialize an array
float times[] = { 12.3, 13.6, 12.0 }; // can leave index blank, C know it is 3
```

```
float sizes[6] = { 52.3, 16.4, 27.8 }; // 6 elements, 3 re initialized
float newscores[4], changescores[4];
float matrix[4][3]; // 2D array

newscores[0] = 76.3;  // first element is 0 – C numbering starts at 0
newscores[1] = 78.3;  // second element is 1
newscores[2] = 49.5;  // third element is 2
newscores[3] = 82.1;  // fourth element is 3

//equating arrays – use a for loop
for(i=0; i<4; i++)
  {
    changescores[i] = scores[3-i];
    printf("changed score = %f\n\r", changescores[i]);
  }
return 0;
}
```

## Functions

You can and will write your own functions. A function can be a short as one line of code. The advantageous of using function are many. A function can have a name that explains itself so that you need fewer comments. A function can be tested thoroughly. Functions can be reused.

A code example with several functions is shown below in Figure 2-11. The first point to note is that functions must be prototyped before the main() function. Typically the functions themselves are defined after the main() function.

```
#include "mcc_generated_files/mcc.h"  // library for SYSTEM_Initialize
#include <stdio.h> // library that has printf()

int QuadraticEquation(int a, int b, int c, int x); //prototype
void MyPrintFunction(int value); //prototype

int main(void) // must have main()!
{
   // Initialize the device – MCC file configures clock and UART
   SYSTEM_Initialize();

   int i = 2, j = 4, k = -2;
   int x = 3, result;
   result = QuadraticEquation(i, j, k, x);
   MyPrintFunction(result);

   return 0;   // successful execution
}

// Evaluate a quadratic equation //
int QuadraticEquation(int a, int b, int c, int x)
{
      int value;
      value = a*x*x + b*x + c;
      return value;
```

```
}

// Print only positive results //
void MyPrintFunction(int value)
{
      if (value >= 0 )
          printf("Result is %i", value);
      else
          printf("Result is negative");
}

Output on display:
Result is 28
```

**Figure 2-11:** A program with user-defined functions.

In C, prototypes must always be defined at the top of the file after directives (e.g. `#include` and `#define` statements). A function can be of datatype `void`, `char`, `int, or float` meaning that it returns nothing, an integer number of size `char`, a integer number of size `int`, or a decimal number of size `float`. The functions may take no variables (`void`), or multiple variables of different datatypes. Note that although a prototype of the form

```
unsigned char IsLeapYear(unsigned int);  //determines if year is a leap year
```

is legitimate C it is not useful. It is much better programming style to include the variable with a meaningful name

```
unsigned char IsLeapYear(unsigned int year);  // No comment needed
```

## Libraries

For easier editing and reusability, the source code can be broken up into separate files. Typically a family of functions is put into the same file. Another file, called a header, will contain the prototypes of those functions. Other files can call those functions as long as the header is included using the `#include` directive. The previous program could be split as follows:

```
#include "myfunctions.h" // for the functions called below

int main(void) // must have main()!
{
      int i = 2, j = 4, k = -2;
      int x = 3, result;

      result = QuadraticEquation(i, j, k, x);
      MyPrintFunction(result);

      return 0;   // successful execution

}
```

**Figure 2-12:** File main.c.

```
int QuadraticEquation(int a, int b, int c, int x); //prototype
void MyPrintFunction(int value); //prototype
```

**Figure 2-13:** File myfunctions.h.

```
#include <stdio.h> // library that has printf()
#include "myfunctions.h"  //prototypes and printf()

// Evaluate a quadratic equation //
int QuadraticEquation(int a, int b, int c, int x)
{
      int value;
      value = a*x*x + b*x + c;
      return value;
}

// Print only positive results //
void MyPrintFunction(int value)
{
      if (value >= 0 )
          printf("Result is %i", value);
      else
          printf("Result is negative");
}
```

**Figure 2-14:** File myfunctions.c.

Note that `stdio.h` is in chevron brackets while `myfunction.h` is in double quotes. The chevron brackets tell the compiler that this file is one of the XC8 compiler's own library of functions.

Remember, well-named functions are a great way to write programs that you or anyone else can understand. You can write the bare bones of a program with all the functions empty. Then you can fill in and test the functions one at a time. For example see Figure 2-15.

```
#include "mcc_generated_files/mcc.h
#include <stdio.h>

void PrintToLCD(string message[]);
void TurnOnLEDonPin21(void);
float ReadADCVoltageOnPin21(void)

int main(void)
{
  // Initialize the device
  SYSTEM_Initialize();

  float voltage;
  PrintToLCD("Hello World");
  TurnOnLEDonPin21();
  voltage = ReadADCVoltageOnPin20();
  printf("Voltage = %f", voltage);
  return 0;
```

```
}

//functions to be defined
```

**Figure 2-15:** Use and name functions to improve readability.

A collection of related functions in a C file and the associated header file is usually called a *library* In group work, large projects are split into smaller amounts with different members of the group assigned to code different libraries. Usually, although we won't be doing it, the C file is compiled to an object file (extension .o) which is not a text file and therefore cannot be read. The header file is simply a text file so it is important that comments and instructions to users of the library be included here. Some brief points

- Headers must end with a blank line. Leave it out and you can get error messages that point to the wrong place in the program usually a line with no problem.
- Header files mostly contain function prototypes but may also have useful #define statements.
- Header functions should not contain #include directives to other libraries.
- The source C file and the header file must have the same name.
- Both the source C file and the header must be added to your project files.
- The source C file must `#include` its own header file before any of the code for the functions . If the functions call other functions from outside that C file, you must also #include the appropriate headers for those libraries. For example if your function in your library calls `printf()`, you must `have #include <stdio.h>` in your source C file.

# Chapter 3. Debugging

The IDE has a debugging feature called the Simulator that allows you to step through lines of code and observe how variables change. In the simulator you can also open a display window to observe printing output. Read *Appendix C* on how to invoke (a) the simulator and (b) the UART display.

## Breakpoints, Watch, and StopWatch

As noted in *Appendix C* , in the simulator you can execute your code line and line and see how the values of variables change, You can even time how long each step takes. Create a project from the file breakpoints.c on the Moodle site. The file has sections labelled Section A, B, and C. Add breakpoints to all the lines in these sections. In *Debugger|Settings* choose the correct operating speed – 1 MHz. Open a stopwatch window from *Debugger|StopWatch*. Open a Watch window from *View|Watch* and add the symbols (i.e. variables) a, b, and c from the drop-down menu. Note that when execution of the program jumps to a breakpoint, the statement on that line has not yet been executed. Any variable value is determined by the lines above the breakpoint.

Use the watch window to view the bit operations of Section A. Step slowly through each calculation and note the value of variables a, b, and c in binary in the Watch window. You can see the values of the variables in decimal, hex, and octal but binary operations make the most sense when the values are shown in binary. When you stop at a breakpoint, that line of code has not yet executed. Recall the logic operation truth tables for & (AND), | (OR), and ^ (XOR – exclusive or). The NOT operator, ~, flips bits so ~1 = 0 and ~0 = 1. Are the results for c consistent with these operations?

| **&** | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

| \| | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

| ^ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

**Figure 3-1:** Truth tables.

We can use the StopWatch window to observe the number of instruction cycles ($T_{CY}$) between two successive breakpoints. For a PIC running at 1 MHz – the default speed of the PIC MCU, $T_{CY}$ = 4/(1 MHz) = $4.0 \times 10^{-6}$ seconds = 4 μs. Use the StopWatch window to record the number of instruction cycles, and the time interval Δt, for the addition of two numbers, first char then int. Which takes longer and why?

Use the StopWatch window to time each delays function and the `NOP()` function. Does the number of instruction cycles match the function call? Why or why not?

## Error and Warning Messages

Most errors are caught by the IDE and a message is given in the output window which often points out on which line the error occurred. For that reason, make sure to turn on the display of line numbers for each C file. Some of the messages are very cryptic and you need to refer to the manual to figure out what the message means. Use the listing error and warning message explanations in *Appendix C* of the *MPLAB XC8 Compiler User's Guide* to debug the following projects.

# Chapter 4. Serial Communication

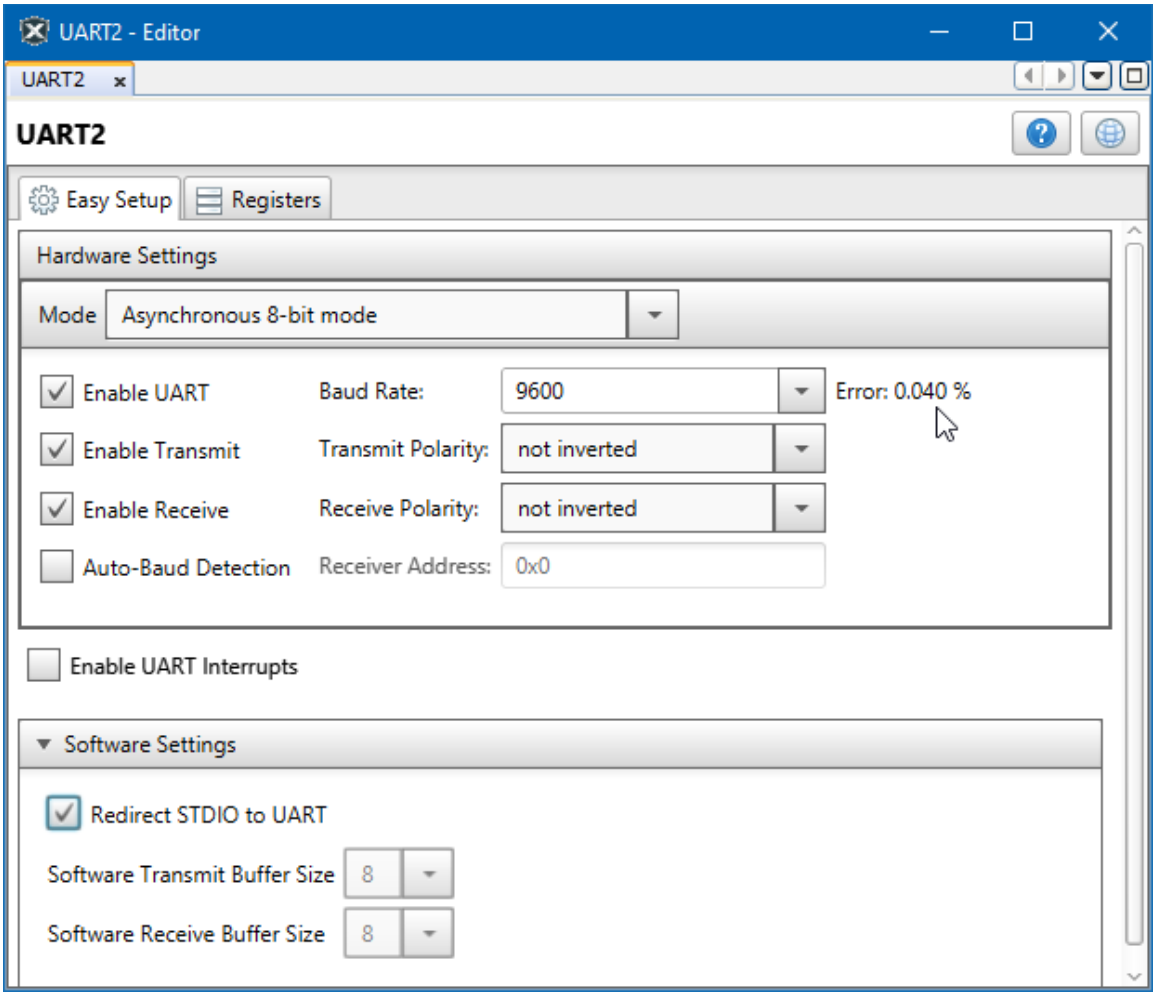
There has always been a need to get two electronic machines to communicate with each other. One of the earliest methods for serial communication is UART (Universal Asynchronous Receiver/Transmitter). UART communication is very common. For example, at one time, every PC has a serial port, labelled COM1, available for serial communication. Today dedicated COM ports on PCs are rare having been passed over in favour of the faster and more flexible USB protocol. The flexible USB protocol can however be used to emulate a COM port. We will be using Serial Communication to allow the PIC MCU to communicate with two different devices, a PC through a USB connection and a small portable display called the SparkFun LCD.

In serial communication, information is sent back and forth in series of ones and zeroes. The ones are represented by a high voltage while the zeroes are at low voltage. To communicate, two machines have same communication speed, character length, parity, and stop bits for proper operation. In serial communication, the speed is called the *baud rate*. The baud rate determines how long a high or low bit lasts; the higher the baud rate the shorter the time. Baud rates of 2400, 9600, 14 400, etc. are typical. Modern UART connectors can go higher, up to 115 200 and more. Character length refers to whether data bits will be sent in groups of eight or nine. Groups of eight are the most common. In the programs you write to communicate with other devices, you first "configure" or choose the settings such as baud rate that you will use for serial communication. The PIC18F46K42 is very flexible in the settings you can use. Many of the devices you will want to connect to will be less flexible and may have only one setting. Therefore, you would program the PIC18F46K42 to match that setting. We will discuss the programming a little later but first let's look at the wiring involved.

The PIC18F46K42 has a two UART peripherals, or built-in circuits, for serial communication. Two pins, TXn and RXn, respectively handle the transmission and reception of serial signals for each module. The n is either 1 or 2 and denotes either the UART1 or UART2 module. Serial Communication is basically timed Digital Input and Output (DIO) (see more in *Chapter 8 Digital Input Output (DIO)*), sequences of on and off signals that each last a specific amount of time. Hence the pin for TXn must be configured as a digital output to send on/off signals and RXn as a digital input to read incoming on/off signals. The receiving device will also have two pins, one TX and one RX. You wire your transmit pin TX to the receive RX pin of the device and your receive pin RX to the transmit TX pin of the device.

To properly configure the UART2 (our default) or UART1 peripherals, we will use the MPLAB Code Configurator (MCC) plug-in for the MPLAB IDE. The *Easy Setup* tab (Figure 4-1 on the next page) asks us to make a small number of choices and it will determine the settings of the Special function Registers (SFRs) to match your choices. You can check the SFRs and their settings in the *Registers* tab.

You will always make the choices shown Figure 4-1 which are *Asynchronous 8-bit mode*, *Enable UART*, *Enable Transmit*, *Enable Receive*, *Redirect STDIO to UART*. We will most often use a




*Baud Rate* of 9600 as it is a common default speed for most devices. There will be times when you may wish to use higher baud rates. Pay attention to the Error percentage beside the Baud Rate. If the error is above 2% there likely will be communication problems. The MCC will notify you that you are exceeding the acceptable error. The Error is the difference between the desired baud rate of 9600 and the actual baud rate (9615) that the PIC/MCU can produce. The baud rates that the PIC/MCU can produce is governed by a formula

$$baudrate = \frac{FOSC}{16 * (spbrg + 1)}$$

where $F_{OSC}$ is the PIC/MCU operating frequency (often 32 MHz) and *spbrg* is an integer value. Since *spbrg* is integral and not continuous, not every baud rate can be achieved. The MCC will determine that *spbrg* = 207 is the best match to your request. The actual baud rate will be

$$baudrate = \frac{32{,}000{,}000}{16*(207+1)} = 9615\ .$$

You must also use the Pin Manager to assign pins to act as RXn and TXn (Figure 4-2 below), For UART2, since it is used to communicate through the USB connection to the IDE, RX2 must be RB7.

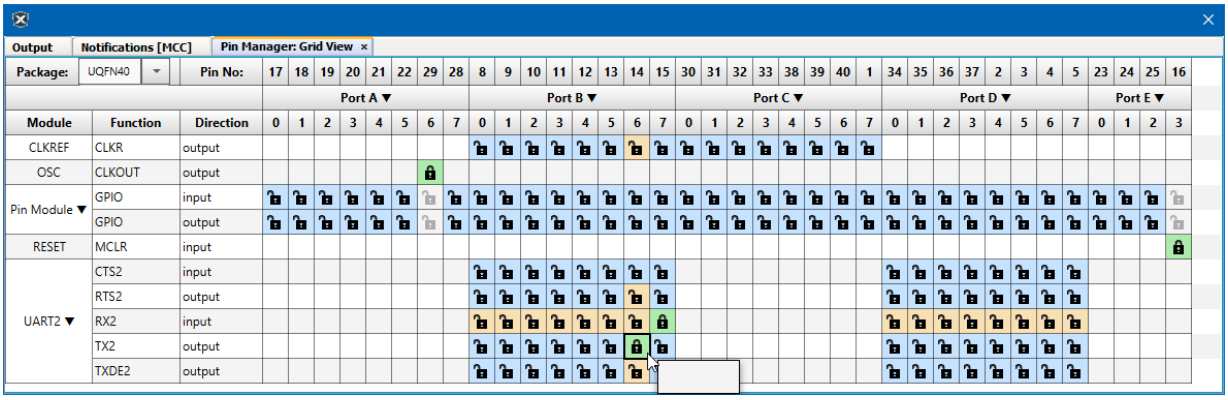



**Figure 4-1 Pin selection for UART2**

Once you have used the Resource Manager to Generate the code, the function SYSTEM_Initialize() will set up now configure the UART peripheral of the PIC/MCU. To send text to the LCD or a terminal window (if properly connected – more on that in *Appendix F PuTTY* and *Appendix G SparkFun* LCD) we could use the `printf()` function from the `stdio.h` library. The code to output a simple message would be

```
#include "mcc_generated_files/mcc.h"  // library for SYSTEM_Initialize()
#include <stdio.h> // library that has printf()

int main(void)
{
  // Initialize the device – MCC file configures clock and UART
  SYSTEM_Initialize();

  printf("Hello world");
  while(1); // wait forever
}
```

## Strings as Input

So far, we have only used the PIC/MCU to write to the terminal screen or to the LCD. We would also like the PIC/MCU to read entries from the terminal screen typed on the computer keyboard. In standard C, in the `stdio.h` library, there is a function `scanf()` similar to `printf()` that does that. However, the XC8 version does not have any such function. There is, however, a function in the `mcc.h` library that can read keyboard entries from the terminal. The function is `UARTx_Read()` which will read a single character. The x stands for 1 or 2 whichever is appropriate for the particular UART peripheral you are using. To read n characters of input we provide you with the `ReadNUART2(string, n)` in the my`UART.h` library which will get a string of up to n characters unless the ENTER key is pressed first.

Reading information that you type into the terminal window on the PC is more complicated than writing to the terminal. When the keyboardist types characters followed by ENTER, the data is sent to a UART buffer on the PIC MCU. The PIC MCU reads from that buffer not directly from the terminal. In a program, you must wait until there is data in the buffer before you read it. You do this using a while loop and a function that checks if there is data waiting in the UART buffer called `UART2_is_rx_ready()` or `UART2_DataReady`. The code would look something like this

```
while(!UART2_DataReady)  // keep waiting for data
      {
         Read data
      }
```

If you use the `UART2_Write(char c)` function of `uart2.h`, you need to similarly make sure that the output buffer is empty before write to it. The `UART2_is_tx_ready()` function performs this task

```
while (!UART2_is_rx_ready())
         {
              UART2_Write('A');
         }
```

When you use the `printf()` function you don't need to worry about this since `printf()` version in the `mcc.h` library does wait for the buffer to be empty before printing the next character.

More on strings in the next chapter.

**Table 4-1:** uart2.h functions

| Function | Description |
|---|---|
| void UART2_Write(char data) | The character to be written to the UART. |
| char UART2_Read(void) | Read a byte (one character) out of the UART receive buffer, including the 9th bit if enabled |
| bool UART2_is_rx_ready(void) | Is date available to be read at RX pin buffer. |
| UART2_DataReady | Identical to UART2_is_rx_ready() |
| bool UART2_is_tx_done(void) | Has transmission finished? |
| bool UART2_is_tx_ready(void); | Is the TX pin buffer busy? Need to wait until it is free to send a message. |

# Chapter 5. Text & Strings

The one byte variable `char` gets its name from the word *character,* that is, a text letter or symbol. It takes a bit of ingenuity to encode symbols using just the zeroes and ones or computer binary memory. The solution was ASCII, a convention for pairing the numbers 0 to 126 with text characters. That still leaves the numbers 127 to 255. These can also be paired with symbols for letters in other Latin script languages like é and Ü. However different machines may have different sets of the upper ASCII characters. A table of the lower ASCII characters is given in *Appendix D*.

You can assign the value to a char variable either numerically or symbolically as shown in the code snippet below. Always use single quotes when assigning a symbol to a variable. The formatting specifier `%c` is used in the `printf()` function to direct the processor to use the ASCII symbol for that number. Note that $ is number 36 in the ASCII table and H is 72.

```
char a, b, c;
int d = 65*256 + 66.
a = 36;    // need ASCII table to know which symbol this is
b = '$';   // use single quotes for characters
c = a + b; // c is ascii character 36 + 36 = 72
printf("a = %c b = %c c = %c\n",a,b,c);

printf("d = %c", d);

Output
a = $ b = $ c = H
d = B
```

Note that `%c` will convert an integer to a byte by ignoring the upper half. The letter B is number 66 in ASCII.

Q. Translate the following integers using the ASCII table. 80 72 89 83 49 54 48 48

The stand C library `ctype.h` , listed in *Appendix H XC8 Libraries*, has a number of useful functions. This includes isdigit(c), which returns TRUE, 1, if character c is '0' to '9' and FALSE, 0, otherwise. Various other functions checks to see if c is alphanumeric, uppercase, lowercase, a space character, and so on. It also has function to raise a lower case letter to upper case and vice versa, e.g. toupper('a') returns 'A'.

## Strings

A string is much like an array of type char except that a string is always terminated by a NULL (0). When you create a string you must always add one to the count of letters in the string. A common mistake is to forget to assign space for this extra NULL character and then bad

unpredictable things can happen. There are several ways to initialize a string as shown in the code below. Note as well the format specifier for string is %s.

```
#include "mcc_generated_files/mcc.h"  // library for SYSTEM_Initialize()
#include <stdio.h> // library that has printf()

int main(void)
{
  // Initialize the device – MCC file configures clock and UART
  SYSTEM_Initialize();

  char name1[4] = "Fred"; // 4 letters + 0 = 5 elements
  char name2[] = "Surname"; // 7 + 0 = 8 elements
  char University[9] = {'K','w','a','t','l','e','n',0};

  printf("Your name is %s %s \n\r", name1, name2);
  printf("Third letter of University is %c \n\r", University[2]);

  while(1); // wait forever
}
```

Working with and manipulating strings is common in C programming. A selection of the functions in the `strings.h` library are described in *Appendix H XC8 Libraries.* Some commonly used functions are `strlen(string)` which calculates the length of the string up to the first NULL character. Another is `strcat(string1, string2)` which concatenates (adds at the end) string2 to string 1. Make sure string1 has enough space! Also you can copy one string into another using `strcpy(string1, string2)`.

Another common programming task is that when we read in a string, say "12.456", that must be converted to a variable of type `float`. The `stdlib.h` library contains a number of functions for converting strings to various number formats including `atof()` for `float`, `atoi()` for `int`, `atol()` for `long`. There are also functions that work the other way taking a number in the various formats and converting it to a string, e.g. ftoa(), itoa(), and `ltoa()`. In the next section there will be some example code showing how these functions work.

## Strings as Input

So far, we have only used the PICMCU to write to the terminal screen or to the LCD. We would also like the PIC/MCU to read entries from the terminal screen typed on the computer keyboard. In standard C, in the `stdio.h` library, there is a function `scanf()` similar to `printf()` that does that. However, the XC8 version does not have any such function. There are, however, functions in the `xc.h` library that can read keyboard entries from the terminal. The functions are `UART2_Read()` which will read a single character. To read n characters of input we provide you with the `ReadNUART(string, n)` in the `myUART.h` library which will get a string of up to n characters unless the ENTER key is pressed first.

Reading information that you type into the terminal window on the PC is more complicated than writing to the terminal. When the keyboardist types characters followed by ENTER, the data is

sent to a UART buffer on the PIC MCU. The PIC MCU reads from that buffer not directly from the terminal. In a program, you must wait until there is data in the buffer before you read it. You do this using a while loop and a function that checks if there is data waiting in the UART buffer called `UART2_DataReady`. The code would look something like this

```
while(!UART2_DataReady)  // keep waiting for data
      {
         Read data
      }
```

A sample program TwoWayCom.c is provided so you can see how this and various functions work.

```
#include "mcc_generated_files/mcc.h"  // library for SYSTEM_Initialize()
#include <stdio.h> // library that has printf()
#include <stdlib.h>   // XC8 Compiler Library for atoi/l/f functiond
#include <string.h>   // XC8 Compiler Library for strcpy() function
#include "myUART.h"   // has Wait_for_ENTER()

//*****************************************************************************
//                               main
//*****************************************************************************
void main (void)
{
  // Initialize the device – MCC file configures clock and UART
  SYSTEM_Initialize();

  unsigned char i = 0, letter;
  char init_string[20]= "Some stored string ";
  char string[20];
//                  1234567890123456789   19 characters plus delimiter
  char blank[20]=  "                    ";
  int number;
  unsigned int big = 50000u;

  // your PIC should be plugged into a USB port
  // PuTTY should be configured

  printf("\n\n\r Serial Communications Test\n\n\r"); \\welcome message
  printf("\n\n\r Press Enter to continue\n\n\r");
  wait_for_ENTER(); \\ waits for you to hit ENTER

while(1)
   {

       strcpy(string,init_string); // put init_string in string
       printf("A stored string - %s\n\r",string);
       printf("Press ENTER to Continue>");
       wait_for_ENTER();

       printf("One character - %c\n\r",'A');
       printf("Press ENTER to Continue>");
       wait_for_ENTER();
       printf("\n\n\rprintf() can also diplay integers in various formats\n\n\r");
       printf("unsigned integer: %u\n\r",big);
       printf("hexidecimal:      %X\n\r",big);
       printf("binary:           %b\n\r",big);
       printf("Press ENTER to Continue>");
```

```
        wait_for_ENTER();

        printf("\n\n\rSingle character input from the keyboard\n\n\r");
        printf("Enter a single character to learns it's ASCII code.\n\r");
        printf("Pressing the ESC key will take you to the next section.\n\r");
        printf(">");

        do {
            while(!UART2_DataReady);   // polls receive buffer for available data
            i = UART2_Read();            // read a single character from buffer
            printf("\n\rASCII code = %i : Symbol is %c\n\r",i,i);
            if ( 27 == i )
               printf("\n\r>\n\rThat was the ESC key ASCII# %d - moving on!\n\r",27);
            else
               if ( isdigit(i) )
                   printf("That was a digit.\n\r");
               else
                  if ( isupper(i) )
                      printf("That was an uppercase letter.\n\r");
                  else
                      if ( islower(i) )
                          printf("That was a lowercase letter.\n\r");
                      else
                          printf("I did not recognize that keypress.\n\r");

            printf(">");
            }
             while ( 27 != i);   \\ ESC key is ascii 27

        // usually input is of variable length and ended by the ENTER (CR) key
        // make sure your strings is at least N+1 in length!
        printf("\n\n\rString input of variable length that ends with
                                          ENTER (CR)\n\n\r");
        printf("Enter a string of up 10 keypresses + CR.\n\r");
        printf(">");
        ReadNUART(string, 11);

        printf("\n\rYour string %s has length %u\n\r",string,strlen(string));

           //You can parse strings into pieces using functions in string.h

        // You can convert strings to numbers

           convert a string to a number
  }
}
```

# Chapter 6. Numbers

A large portion of programming involves simple arithmetic operations such as $c = a + b$. We are so used to simple addition and other operations that we sometimes forget how many years it took for us to master the concepts involved. You are so accomplished at it now that if I say that *a* is 10 and *b* is 5, you will quickly compute that *c* is 15. You would not be so fast if I ask for the sum of 178786485 and 6211456. Without a calculator you will probably revert to pen and paper using the method you learned in grade school – sum the digits, and if the sum is greater than 9 carry the 1 (see Figure 6-1).

```
    1     11
  178786485
+   6211456
-----------
  184997941
```

**Figure 6-1:** Elementary addition.

There are similarities between how numbers are added in the above example and how programs and processors handle the same operation. Like you, the program needs space to write the given numbers, the answer, and intermediate steps like carrying the one. It does not use scratch paper, of course, but solid-state memory instead. Unlike you it does not use decimal, base 10, numbers but rather binary, base 2, numbers – the zeroes and ones of digital logic.

All RAM "data" is accessed and saved in the PIC MCU in 8-bit bytes. One-bit access and saving are also possible. To access multiple bytes, the PIC must do sequential read or writes. A C (or C++) compiler will set aside the correct number of storage locations needed for a particular variable type when a variable is defined. Common variable keywords are `char`, `int`, and `long`, which indicate variables that are stored in one, two, and four bytes, respectively. Note that although `char` is short for character, it is in fact a number type as shown in the code snippet below.

```
char a = 10;
char b = 5;
char c;
c = a + b;
```

A `char` variable has one byte, or eight bits, and each bit can either be a zero or a one. The possible values of that variable are 00000000 to 11111111. By convention we break the set of eight bits into two sets of four bits called *nybbles* for easier reading, e.g. 0000 0000 and 1111 1111. With eight bits there are exactly $2^8 = 256$ possible numbers. By the same logic, an `int` can hold exactly $2^{16} = 65{,}536$ numbers while a `long` can hold $2^{32} = 4{,}294{,}967{,}296$ numbers. There is a further categorization of these integer types into `unsigned` and `signed`. An `unsigned` variable type contains only positive integers and zero, while the `signed` variable type can hold negative numbers

as well. When you leave out `unsigned` or `signed` in the variable declaration, the default is `signed`. So in the code snippet above, the variables *a*, *b*, and *c* could hold negative numbers if we wished.

With unsigned variables there is a simple relationship between the memory space and the number it represents or holds. For example, in a `char`, 0000 0000 = 0, 0000 0001 = 1, 0000 0010 = 2, and so on up to 11111111 = 255. The relationship between memory and signed numbers is more complicated and will be discussed later.

## Overflow

One point to keep in mind is that the memory space for a variable is fixed. Consider the code snippet below,

```
unsigned char a = 100;
unsigned char b = 200;
unsigned char c;
c = a + b;
```

In an ideal world, we would have the result that *c* = 300. But 300 is bigger than 255. In binary 300 = 1 0010 1100 but this is nine bits long. There is no room for the 9$^{th}$ bit to be stored in the allotted memory space so it disappears and we are left with *c* = 0010 1100 = 44. Memory space is like a loop or ring going from 0000 0000 to 1111 1111 in the case of a `char` variable. When you add 1 to 1111 1111 you are back to 00000000. This happens on mechanical counters, like your car odometer, as well. This rollover or overflow is a common source of programming bugs. You need to make sure that there is sufficient memory space for the calculations your program will do.

We don't need to convert 300 to binary to see what number a computer will treat it as. Simply note that 300 = 1 × 256 + 44. This is equivalent to subtracting 256 ($2^8$) as often as necessary until the remainder is less than 256. What number would `unsigned char a = 1200` be?

Q. Consider the following `unsigned int` variables. What numbers are they equivalent to?

```
unsigned int a = 80000;
unsigned int b = 1000000;
```

## Storing Negative Numbers

Since you cannot store the negative sign in the memory space (it can only hold ones and zeroes), storing negative numbers is a bit of a challenge. One logical and efficient solution starts by taking 00000000 as zero and as the centre of the ring of storage space. The first number above zero in the ring would be 00000001 and we say that is decimal 1. Similarly, the first number below zero in the ring would be 11111111 and we call that –1. Continuing in the same fashion we stop when the two numbers would overlap. The series is illustrated in the table below Table 6-1. One advantage

of this approach is that negative numbers are easy to identify because the 8th bit (or bit 7) of a negative number is 1.

| 10000000 | 10000001 | … | 11111110 | 11111111 | 00000000 | 00000001 | 00000010 | … | 01111110 | 01111111 |
|---|---|---|---|---|---|---|---|---|---|---|
| –128 | –127 | … | –2 | –1 | 0 | 1 | 2 | … | 126 | 127 |
| 0x80 | 0x81 | | 0xfe | 0xff | 0x00 | 0x01 | 0x02 | | 0x7e | 0x7f |

**Table 6-1.** Storing negative and positive numbers in memory.

For a `char` variable, signed values run from –128 to 127. For an `int`, signed values run from –32,768 to 32,767 and for a `long` from –2,147,483,648 to 2,147,483,647.

Rollover will also happen with signed variables. For example, `signed int a = 128` is equivalent to –128 and `signed int b = -129` is equivalent to 127. In general, if you wish to know the actual value of unsigned char number either subtract or add 256 ($2^8$) until the remainder is in the range –128 to 127. For instance, `signed int c = 200` is equivalent to –56 since 200 –256 = –56 and `signed int d = -250` is equivalent to 6 since –250 + 256 = 6.

Q. Consider the following `signed int` variables. What numbers are they equivalent to?

```
unsigned int a = -80000;
unsigned int b = 120000;
```

## Printing Numbers in Different Formats

We are most used to writing numbers in decimal notation, e.g. 125. There are other formats with different bases. In base-2 or binary, 125 would be written 0b01111101. The '0b' prefix is used to indicate that we are using binary notation. Note that leading zeroes can be dropped, 2 = 0b00000010 = 0b10. Another common format is hexadecimal, or base-16, format. In base 16, the letters a, b, c, d, e, and f are used for the numbers 10 to 15 respectively. In hexadecimal, 125 = 0x7d. Note that 125 = 7*16 + d = 112 + 13. Again, a prefix, '0x', identifies hexadecimal format.

In your code you can write any integer number in any of these formats, like the code snippet below, where *a* = *b* = *c*. Each number is stored exactly the same in memory.

```
unsigned char a = 125;
signed char b = 0b11111011;
unsigned char c = 0x7d;
```

The `printf()` function will display a number of size `char` or `int`, one or two bytes, in decimal, octal, or hexadecimal formats using the formatting specifiers `%u` for unsigned integer, %i or `%d` for signed decimal, %o for octal, and `%x` for hexadecimal such as

```
printf("Various formats: a = %x b = %i c = %u c = %o.", a, b, c, c);
Output
Various formats: a = 7d b = -5 c = 125 c = 175.
```

Note that the hexadecimal and octal output may look like decimal output. You may wish to do the following to make the base clear.

```
printf("Various formats: a = 0x%x b = %i c = %u c = o%o.", a, b, c, c);

Output
Various formats: a = 0x7d b = -5 c = 125 c = o175.
```

Unfortunately, XC8 (unlike some C Compilers) does not support a binary %b format specifier.

Please note that the `printf()` function allows a great deal of flexibility in what you can print but the downside is that there is limited error checking and if you do not match the formatting to variable type, the results may not make sense as demonstrated in the code snippet below.

```
char a = 0b11111111;          // 255 = -128
unsigned char b = 0b11111111; // 255
printf("Unsigned a = %u - Signed a = %i\n", a, a);
printf("Unsigned b = %u - Signed b = %i\n", b, b);

Output
Unsigned a = 65355 - Signed a = -1
Unsigned b = 255 - Signed b = 255
```

For numbers of size `long`, 4 bytes, use the formatting specifiers `%lu` for unsigned decimal, `%ld` for signed decimal, and `%lx` for hexadecimal rather than `%u`, `%d`, or `%x`. As mentioned, `%u`, and `%x` work with two bytes not the four of a `long` number. The `printf()` function does not check to see if the variable agrees in size with the formatting specifier with interesting results. As shown in the code snippet below, the `printf("%u", longnumber)` only prints the lower two bytes. This is equivalent to the remainder after dividing the number by $2^{16}$ = 65536. You can use `%u` twice as in `printf("lower: %u upper: %u", longnumber)` but then you get the remainder and number of times 65536 goes into the value. You can use the hex specifiers twice to get the right answer (although in the wrong order) but you need to keep leading zeroes in the lower byte, 4 for hex (see the code example for details).

```
unsigned long k = 723458; // = 0b10110000101000000010 = 0xb0a02;
// the problem with using a too small formatting specifier
printf("k (base 65536) = %u\n",k); //%u only reads lower two bytes of k
                                   // hence it rounds k in base 65536
printf("k = %u + %u*65536 \n",k); //reading both lower and upper bytes
                                  // remainder + k/65536
printf("k = %04x %x\n",k);        //reading both lowerr and upper bytes

// correctly specifying the size
printf("k = %lu = %lx\n", k, k);
```

```
Output
k (base 65536) = 2562
k = 2562 + 11*65536
k = 0a02 b

k = 723458 = b0a02
```

Q. What would `printf("%x%x %b%b", k, k)` display? What are the decimal equivalent values of these numbers as displayed?

## Decimal or Floating-Point Numbers

Decimal numbers such as 0.0032 and $2.63 \times 10^4$ require a lot more processing to store in memory that integers. Arithmetic operations involving decimal number are also more complicated than those involving only integers. Decimal number have type `float` or `double`.

If you use a literal number (e.g. 50000) rather than a variable with the `printf()` function you will also run into problems because C assumes that any literal without a decimal in it is an `unsigned int`. The *cast* operator is one way of explicitly making the correct amount of memory space available although it is typically used with variables. The simplest method to indicate that the literal is unsigned is by adding a `u` at the end of the number. Similarly, adding a terminal `l` or `ul` indicates that the number is `signed long` or `unsigned long`.

```
printf("%i\n",50000);  // Must be very careful with numbers bigger than
                       // 32767
                       // Compiler expects numbers to be signed integers
                       // which go from -32768 to 32767.
                       // Doesn't know how to handle this number. It treats
                       // it as a char - size one byte. But 50000 is two
                       // bytes big. In binary 50000 = 11000011 0101000.
                       // Compiler ignores LSB (second byte) and assumes
                       // number is 11000011 or 195.
                       // Not all C compilers do this.

printf("%i\n",(unsigned int)50000); // the cast operation lets the compiler
                       // understand the size but converts it to a signed
                       // integer since use %i

printf("%u ",(unsigned int)50000);

printf("%i ", 50000u); //terminal u indicates unsigned int but %i converts
                       //value to signed int
printf("%u ", 50000u); //matching types



Output
195
-15536
50000
-15536
```

```
50000
```

You have your choice of printing a decimal number in ordinary or scientific notation using %f or %e.

```
printf("%f\n",12300.5250000);
printf("%12.5e\n",12300.5250000);  // Sci notation to 5 decimal places.


Output
123000.525
1.23001e+4
```

## Arithmetic and Memory

Simple arithmetic can be the source of an amazing number of programming mistakes. This is largely because we forget the limitations of the memory space assigned to numbers. For example, consider the code snippet below.

```
int a = 8;
int b = 52;
int c = 100;
int d;

d = a/b*c;
printf("%i",d);

Output
0
```

We are usually surprised at the result because we know that 8/52 × 100 = 15.38 which when rounded to an integer should give 15. However, the first operation, 8 ÷ 52, is between integers and must go in a space for an integer. The value 0.1538 is thus rounded to zero before being multiplied by 100 yielding 0. You can make room for the calculation by using a *cast* operation, but you need to use care in doing so as shown below

```
int a = 8;
int b = 52;
int c = 100;
int d;

d = (float)(a/b)*c;  // cast mis-applied
printf("%i\n",d);
d = (float)a/b*c;  // correct cast
printf("%i",d);

Output
0
15
```

The first cast operation of a/b to `float` occurs after the division and is thus too late. The variable `a` must be promoted to `float` first. A `float` divided by an `int` is of size `float`. A `float` multiplied by an `int` is also a `float`. The final step is the rounding to an `int` to get the value of `d`.

Q. What will the following code produce on output? How would you change the code to produce more sensible results?

```
unsigned char a = 70;
unsigned char b = 100;
int c = 1000;
int d;

d = a*b – c;
printf(“d = %i\n”,d);
```

## Text and Memory - ASCII

The one-byte variable `char` gets its name from the word *character,* that is, a text letter or symbol. Just as it required a bit of ingenuity to encode for negative numbers with just zeroes and ones, the same holds for text. The solution was ASCII, a convention for pairing the numbers 0 to 126 with text characters. That still leaves the numbers 127 to 255. These can also be paired with symbols for letters in other Latin script languages like é and Ü. However different machines may have different sets of the upper ASCII characters. A table of the lower ASCII characters is given in *Appendix D*.

You can assign the value to a char variable either numerically or symbolically as shown in the code snippet below. The formatting specifier `%c` is used in the `printf()` function to direct the processor to use the ASCII symbol for that number. Note that $ is number 36 in the ASCII table and H is 72.

```
char a, b, c;
int d = 65*256 + 66.
a = 36;
b = ‘$’;
c = a + b;
printf("a = %c b = %c c = %c\n",a,b,c);

printf(“d = %c”, d);

Output
a = $ b = $ c = H
d = B
```

Note that `%c` will convert an integer to a byte by ignoring the upper half. The letter B is number 66 in ASCII.

Q. Translate the following integers using the ASCII table. 65 80 83 67 49 50 57 57

## Bitwise Operators

C has a set of operators that derive from Boolean logic. They are *and*, *or*, *exclusive or* and *not* (&, |, ^, and ~). Note that the number 1 is synonymous with true and 0 with false. The truth tables for the first three operators are

| & | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

| \| | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

| ^ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

**Table 6-2:** Truth tables.

The not operator works on individual bits so ~1 = 0 and ~0 = 1. The code snippet below reproduces these results. The results would be much easier to understand if the variables were written in binary.

```
char a = 0b0011, b = 0b0101, c;

c = a & b;
printf("a = %u & b = %u => c = %u\n\r",a,b,c);
c = a | b;
printf("a = %u | b = %u => c = %u\n\r",a,b,c);
c = a ^ b;
printf("a = %u ^ b = %u => c = %u\n\r",a,b,c);

a = ~a;
b = -b;
printf("not a = %u and not b = %u\n\r", a,b);

Output
a = 3 & b = 5 => c = 1
a = 3 | b = 5 => c = 7
a = 3 ^ b = 5 => c = 6
not a = 252 and not b = 251
```

Two other operators that work at the bit level are the left shift << and right shift >> operators. The usage is clearest on integers in binary format. If c = 0b00101 then c = c << 2 yields c = 0b0010100. Everything moved left two places and the new positions are filled with zeroes. Similarly if c = 0b00101, then c = c >> 2 moves everything right two places. This yields c = 0b00001. Shift operators are equivalent to $c = c / 2^n$ (right shift n places) and $c = c * 2^n$ (left shift n places). The shift operations are much faster than the equivalent expressions for division or multiplication.

Be careful using shift operators on large signed numbers though.

```
int a = -32000, b = 32000, c;

c = a << 2;
printf("a = %i, c = %i\n",a,c);

c = b >> 2;
```

```
printf("b = %i, c = %i",b,c);

Output
a = -32000, c = 8384
b = 32000, c = -3072
```

Q. What is `unsigned int a = (1234 & 724) << 2`?

## Bit Masks

Many of the functions, like DIO and ADC, of the MCU are controlled by a single bit of special bytes of computer memory called Special Functions Registers (SFRs). The bit often turns a particular aspect on (1) or off (0). Since the smallest quantity C can handle is the byte, bit masks are a way to read the current setting of a particular bit or to set a particular bit.

If we have `unsigned char` variable named `sfr` and two more variables `mask_bit5 = 0b0010 0000` and `mask_bit6 = 0b0100 0000`, then we can *set* bit 5 (make it have value 1) of `sfr` with the code

```
sfr = sfr | mask_bit5
```

or more compactly

```
sfr |= mask_bit5.
```

We can even set two bits at once with

```
sfr = sfr | mask_bit5 | mask_bit6
```

If we wish to *clear* bit 5 (make it have value 0) of `sfr`, we use the code

```
sfr = sfr & ~mask_bit5
```

or more compactly

```
sfr &= ~mask_bit5
```

We can also clear multiple bits at once

```
sfr = sfr & ~mask_bit5 & ~mask_bit6
```

We can check if bit 5 is set using

```
(sfr & mask_bit5) == mask_bit5
```

That is equivalent to the shorter

```
sfr & mask_bit5
```

Examining the identifiers above we see that bit 7 set the transmit interrupt on or off. Similarly, every other bit is controlled by a pair of identifiers. Rather than remember which bit does what, the programmer ANDs the appropriate choices together. The bit patterns that are all 1s will have no effect when ANDed with the others but should be included for clarity.

Q. Consider the literal bit mask in the code example at the bottom of Figure III-2. Using the macro definitions, what is the 8 bit result?

## Structures and Bitfields

In C, there are a number of basic variable types such as `int` and `char`. C also allows the programmer to define new variable types and declare variables of the new type. A new variable type can be created by grouping basic types together. This grouping is called a *structure*. For example, we could define a structure for a Student ID that combines a last name (a string variable) with a student number (a long integer).

```
typedef struct studentID {
  char studentname[20];
  long studentnumber;
}
```

Then you can create variables of this type

```
studentID PHYSStudents[70];
```

You can now access the fields of the variable (`studentname` and `studentnumber`) using the *member of* operator which is a period ".".

```
strcpy(PHYSStudents[1].studentname, "Smith");
PHYSStudents[1].studentnumber = 10123456;
```

One can also create a structure that is named groups of bits (such a structure called a *bitfield*) or create a structure that is just named individual bits (this is also a structure called a bitfield). Microchip has done so for the Special Functions Registers (SFRs) and these variables are already defined in `xc.h` and make working with the SFRs much simpler.

### Unions

Unions allow you to save space by stuffing more than one variable in the same memory. Of course, it must make sense to do. Since we often deal with a set of bits, unions can let us deal with them as a set or individually. Consider the following structure and union

```
struct setLED
{
    unsigned  Zero:1;
    unsigned  One:1;
    unsigned  Two:1;
    unsigned  Three:1;
```

```
    unsigned  Four:1;
    unsigned  Five:1;
    unsigned  Six:1;
    unsigned  Seven:1;

};

union sensor_union
{
    unsigned char Byte;
    struct setLEDs bits;
};
```

The structure could be used with a set of eight indicator LEDs. The zeroth bit, controlling the first LED, is labelled Zero and the seventh bit is labelled Seven. The union lets us work with the bits individually or as a group of size `char`.

A statement

```
union sensor_union IndicatorLEDs.Byte = 0; //create and initialize union
```

creates the space for the information and initializes the values to zero. We can set the values we want each LED to have using

```
indicatorLEDs.bits.Zero = 1;
indicatorLEDs.bits.One = 0;
indicatorLEDs.bits.Two = 0;
indicatorLEDs.bits.Three = 1;
indicatorLEDs.bits.Four = 1;
indicatorLEDs.bits.Five = 0;
indicatorLEDs.bits.Six = 1;
indicatorLEDs.bits.Seven = 1;
```

Then

```
printf("%b", indicatorLEDs.Byte);
```

will output `0b11011001`. We could reverse this, set all the bits at once using

```
indicatorLEDs.Byte = 0b00000001;
```

and now

```
printf("%" indicatorLEDs.bits.Zero);
```

will output 1.

## Random Numbers

It is common to want to generate random numbers to model events like drawing a card from a deck of playing cards. It is difficult to actually generate random numbers in code, the closest we get is called a pseudo-random number. It looks random until you look at the statistics of a large group of them and then you may be able to predict the next number in the series which is impossible for truly random numbers. In the `stdlib.h` library there is the function `rand()` that returns a integer number between 32767. The only trouble with this function is that every time a program runs, it will give the same sequence of "random" numbers. That may be useful for debugging but is usually a bad thing. A second function to get around this is `srand(int seed)`, which causes the `rand()` function to start at other values. However, if `srand(seed)` in initialized use the same seed each time the program runs, the sequence of random numbers will be the same. So, to have a different random number sequence for every run takes a little trick, you need to initialize `srand(seed)` with a seed that is different between program runs. One way to do this is to use a Timer (see *Chapter 10 Timers*). The Timer clock counts clock ticks very quickly with a counter that repeats once it reaches 65535. If you ask for user input, the user input will vary from program run to program run based on user reaction times. When the user input comes in, call `srand(counter value)` once and the random number sequences will appear to vary from run to run (actually you will only have 65536 different sequences). Example code is shown below.

```
#include "mcc_generated_files/mcc.h"  // library for SYSTEM_Initialize()
#include <stdio.h> // library that has printf()
such as printf()
#include <stdlib.h>   // XC8 Compiler Library for srand() and rand() functions
#include "myUART.h"   //  wait_for_ENTER()

int main (void)
{
  // Initialize the device - MCC file configures clock and UART
  // TMR0 maximum period with prescaler N = 2
  SYSTEM_Initialize();

  int number;

  printf("%c%c%c%c",27,'[','2','J'); // Clear PuTTY Window
  printf("%c%c%c",27,'[','H');       // Home PuTTY Cursor

  printf("Random Number Test\n\n\r"); //welcome message

  printf("Press Enter to start> ");
  wait_for_ENTER(); // waits for you to hit ENTER (in configureUART library)
  srand(TMR0_Read());  // we won't know TIMNER value b/c user hitting enter will
                       // vary omit this line & always get same sequence

   while(1)
      {
          number = rand();
          printf("Your random number is %u     \n\r",number);
          printf("Normalized to 0 to 51: number is %u\n\n\r",number % 52);
          printf("Press Enter to continue> ");
          wait_for_ENTER(); // waits for you to hit ENTER
      }
 return 0;
}
```

**Figure 6-2:** Generating a random number

Try commenting out the `srand()` line in the code above and restarting the program several times to see what happens. By the way, it is possible to generate random numbers from electronic circuits but we won't do that.

# Chapter 7. Review of Electric Circuits

## Current

Electrons are negatively charged particles. The movement of electrons constitutes the flow of *electric current*. The movement of a negative charge in one direction can be thought of as the movement of a positive charge in the opposite direction. The direction of the positive charge is the direction of *current flow*. You can think of an electric charge as water and of an electric current as water flowing in a river or a pipe. The water flows in a certain direction – from the higher ground to the lower ground. Electric current flows from high potential (voltage) to low. The measure of current is *ampere* (A)**.** An instrument called *ammeter* is used to measure the current that passes through it.

## Battery and Voltage

You can think of an electric battery as a generator of a positive charge. Once an electric load is connected to the battery, the current flows from its positive terminal (+) to its negative terminal (–) through the connecting wires and the load. The batteries come with different voltage ratings expressed in *Volts* (V). For instance, a car battery is a 12-volt battery. The higher the voltage, the more current will flow from the battery when you connect an electric load to it. You can think of a battery as a pump, which pumps the water from the lower ground (usually referenced as zero) to a higher ground. You can think of a battery's voltage as the height of the elevated water level. The battery is often called a *voltage source*. Voltages are measured by an instrument called *voltmeter*. To measure the voltage, the voltmeter must be connected between two points where there is a certain voltage difference (like between the + and the – terminals of the battery). The reference point used to measure voltages is called *ground*. For example, one of the terminals of the car battery is connected to the car's body and is considered an electric ground. Battery and associated symbols are shown in Figure 7-1.

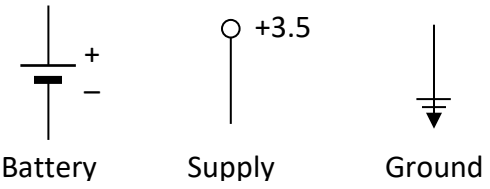



**Figure 7-1:** Battery, power supply, and ground symbols.

## Resistors

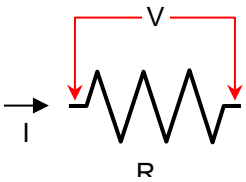



**Figure 7-2: Ohm's Law.**

The basic element of electrical and electronic circuits is the resistor. The resistor obeys *Ohm's Law*

$$V = IR \qquad [\ 7\text{-}1\ ]$$

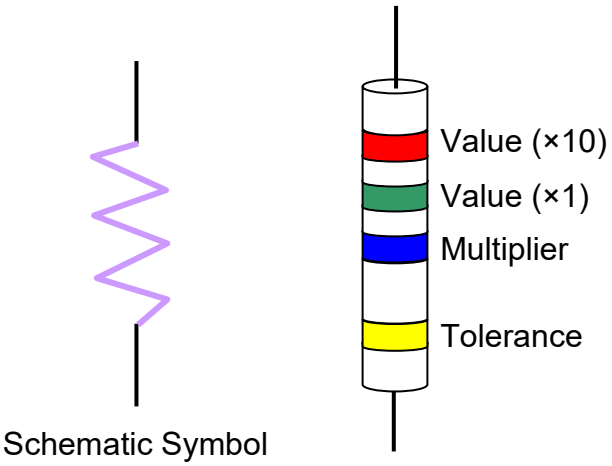
where $V$ is the voltage drop across the resistor, $I$ is the current through the resistor, and $R$ is the resistance of the resistor (see Figure 7-2). Resistors dissipate energy as heat (this is known as *Joule Heating*) according to the formula

$$P = IV = I^2R = V^2/R. \qquad [\ 7\text{-}2\ ]$$

Resistors are rated by how much heat they can dissipate without melting, e.g. ¼W.

## Colour Code

Resistors come in a wide variety of standard resistances. The value of the resistance is give by the colour bands on the resistor. One band isolated from the other three gives the tolerance or accuracy of the resistance value. The other three bands encode the value of the resistance. The colour code is given in the diagram Figure 7-3 below.




| Colour | Value | Multiplier |
|---|---|---|
| Black | 0 | $10^0$ |
| Brown | 1 | $10^1$ |
| Red | 2 | $10^2$ |
| Orange | 3 | $10^3$ |
| Yellow | 4 | $10^4$ |
| Green | 5 | $10^5$ |
| Blue | 6 | $10^6$ |
| Violet | 7 | $10^7$ |
| Gray | 8 | $10^8$ |
| White | 9 | $10^9$ |

| Colour | Tolerance |
|---|---|
| Gold | 5% |
| Silver | 10% |
| None | 20% |

**Figure 7-3:** Symbol and colour code for resistors.

As an example, a resistor with red, green, and blue bands in sequence separated from a gold band would have a resistance of $R = (2 \times 10 + 5 \times 1) \times 10^6\ \Omega \pm 5\% = 25 \times 10^6\ \Omega \pm 5\%$.

Q. What is the resistance of a resistor with yellow, violet, brown and silver bands?

## Series/Parallel Connections

Resistors can be connected by conducting wires (or paths) in series, or in parallel, as shown in the schematic diagram Figure 7-4 (a) and (b) where the symbols for battery and resistors are used.

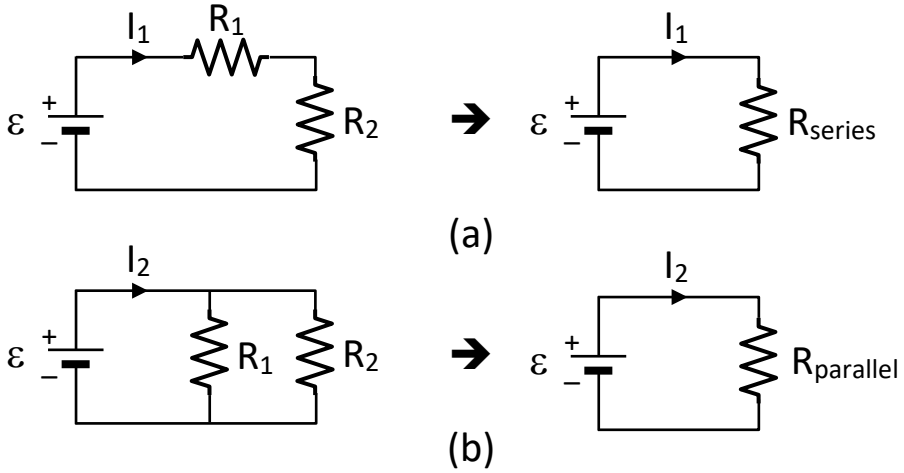



**Figure 7-4:** (a) Series and (b) parallel connections.

The equivalent resistors will draw the same current *I* from the battery. Their value can be calculated as:

$$R_{series} = R_1 + R_2 \qquad [\ 7\text{-}3\ ]$$

$$R_{parallel} = R_1 \,||\, R_2 = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2}}. \qquad [\ 7\text{-}4\ ]$$

Note that series resistors always carry the same current.

## Shorting

One consequence of Eqn [IV-4], is that the equivalent resistance of two resistors in parallel is always less than either of the two resistors. Since parallel resistors always have the same voltage drop, $V_1 = V_2$. Ohm's Law, applied to this result, means $I_1R_1 = I_2R_2$. So the smaller resistor carries the bigger current. These results reach their extreme conclusion when a resistor is *shorted*. This means a wire of negligible resistance is connected across a resistor as shown in Figure 7-5 below. Having no resistance, all the current flows through the wire and no current flows through $R_2$. The equivalent resistance of $R_2$ and the wire is zero, so they can be replaced by the wire itself as shown in Figure 7-5. Shorting resistors unintentionally can cause big problems. If $R_1$ is a small resistor, the current produced by the battery could be dangerously high, the power rating of the resistor could be exceeded and the Joule heating could melt the resistor or even start a fire.

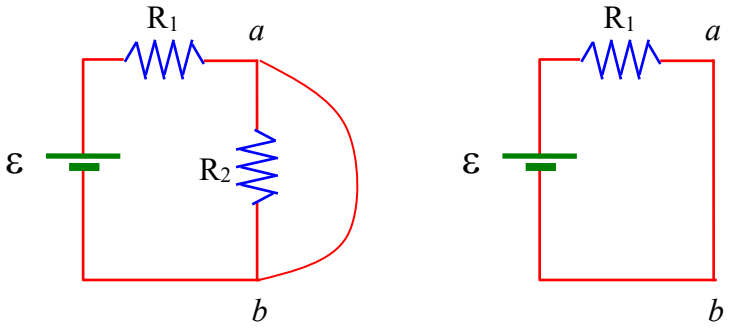



**Figure 7-5:** A short circuit.

## BreadBoard

A breadboard is a convenient way to assemble circuits where the parts are small such as microchips, diodes, and resistors. Pinholes in the front of the breadboard let you securely connect circuit elements to it. The sets of pinholes are connected by conducting bars in two different patterns, short horizontal and long vertical, under the plastic cover, see Figure 7-6. This setup allows parallel and series arrangements of elements to be put together quickly. When using wire connectors make sure to keep them short and neat!

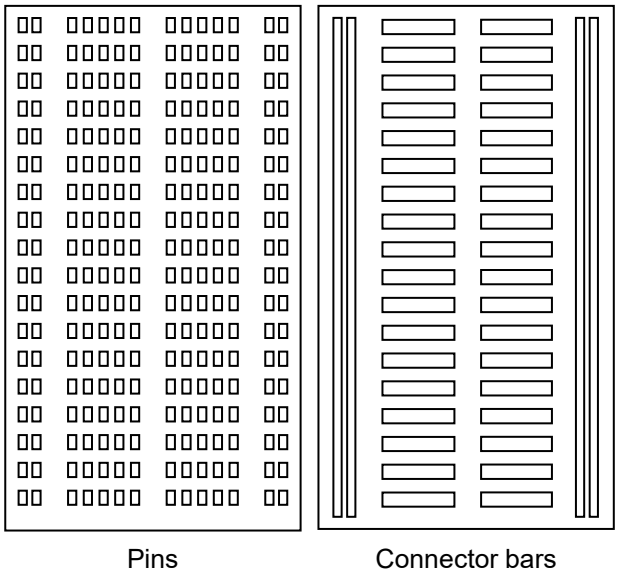



**Figure 7-6:** Breadboard connectors.

## Electronics Style Circuit Diagrams

In electronics, batteries and other power supplies are seldom shown. A small circle indicates where one end of the power supply goes. The other end is assumed to be connect to ground (symbol ⏚). There may be more than one ground symbol in a diagram. This indicates that a wire connecting the two points has not been drawn. The diagram below Figure 7-7 illustrates how a standard circuit could be drawn in this style. In general, we will use the electronics style from now on.

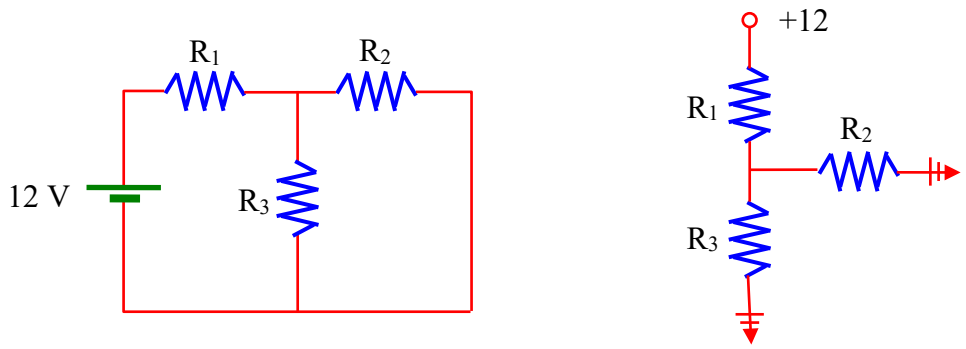



**Figure 7-7:** Standard and electronics style circuit diagrams.

## Kirchhoff's Rules

More complicated circuits are usually analyzed using *Kirchhoff's Rules*. Consider the circuit in Figure 7-8 below. Points *b* and *e*, where three or more wires meet, are known as *nodes* or *junctions*. A *branch* is a path from one node to another. In Figure 7-8 there are three branches, *be*, *bcde*, and

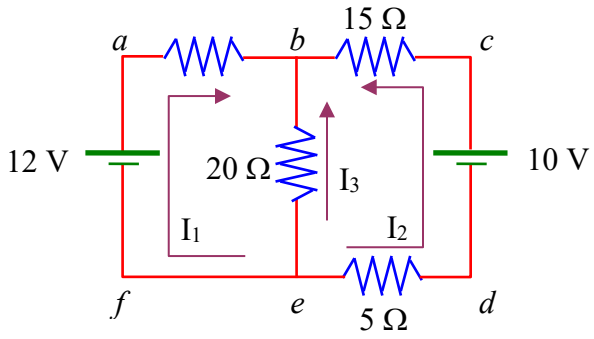



**Figure 7-8:** A complicated circuit.

*bafe*. These branches are said to be parallel since the start and end nodes are the same. Each branch carries a single current. Elements in the same branch are said to be in series, so the 5 Ω and 15 Ω resistors in branch *bcde* are in series. At a node, *Kirchhoff's Current Rule* is obeyed,

$$\sum I_{in} = \sum I_{out} , \qquad [\ 7\text{-}5\ ]$$

the sum of the currents entering a node equals the sum of the currents leaving a node. For the nodes, we have

$$I_1 + I_2 + I_3 = 0 \quad \text{(point } b\text{)}$$

and

$$0 = I_1 + I_2 + I_3 \quad \text{(point } e\text{)}.$$

As in the example above, the equations do not have to be independent.

Kirchhoff's Current Rule is simply conservation of charge – electrons are neither being created not destroyed in the circuit.

A *loop* is a closed path from a point back to itself. Where you start is arbitrary. Several loops in the circuit in Figure 7-8 are *fabef* and *dcbed*. Around a loop, *Kirchhoff's Loop Rule*:

$$\sum_{loop,\, i} V_i = 0, \qquad [\ 7\text{-}6\ ]$$

is obeyed where $V_i$ is the voltage drop across each circuit element in the loop. The orientation of the voltmeter must be kept the same as you go around the elements of the loop. For loop *fabef* we get

$$(12\ V) - (10\ \Omega)I_1 + (20\ \Omega)I_3 = 0 \qquad \text{(loop } fabef\text{)}$$

and for loop *dcbed* we find

$$(10\ V) - (15\ \Omega)I_2 + (20\ \Omega)I_3 - (5\ \Omega)I_2 = 0. \quad \text{(loop } dcbed\text{)}$$

Note that battery voltages are positive when we travel from the negative to the positive side and vice versa. The voltage drop across a resistor is negative if we travel around the resistor in the direction of the current through it and positive if the current is in the opposite direction.

Kirchhoff's Loop Rule is simply conservation of energy since voltage is energy per charge.

Solution of the equations is straightforward, if tedious, as long as you have as many independent equations as you have unknown currents.

Applying Kirchhoff's Laws to circuits written in electronics style is somewhat simpler. As before, at the nodes apply circuits current laws. Note that even if a branch has no resistors in it, you still assign a current as between points *e* and *f* in Figure 7-9.

Here the node or current equations are

$$I_1 + I_2 + I_3 = 0 \quad \text{(point } b\text{)}$$

and

$$0 = I_3 + I_4 + I_5 \quad \text{(point } e\text{)}.$$

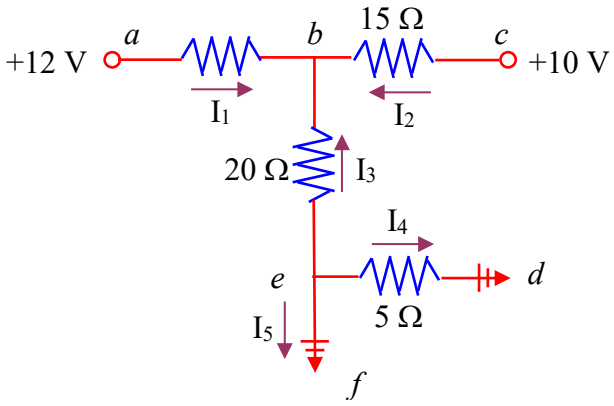



**Figure 7-9:** A complicated circuit in electronics style.

In electronic style diagrams you do not appear to have loops. But you do know the voltage drop between each battery point and each ground point.

Here, we can go from a battery to ground four different ways and we get the equations:

$$(12\ V) - (10\ \Omega)I_1 + (20\ \Omega)I_3 = 0 \qquad (a \to f)$$

$$(12\ V) - (10\ \Omega)I_1 + (20\ \Omega)I_3 - (5\ \Omega)I_4 = 0 \qquad (a \to d)$$

$$(10\ V) - (15\ \Omega)I_2 + (20\ \Omega)I_3 = 0 \qquad (a \to f)$$

and

$$(10\ V) - (15\ \Omega)I_2 + (20\ \Omega)I_3 - (5\ \Omega)I_4 = 0 \qquad (a \to d).$$

Again, you have more equations than unknowns.

## Voltage Divider

Shown in Figure 7-10 is a simple circuit called the *voltage divider*.

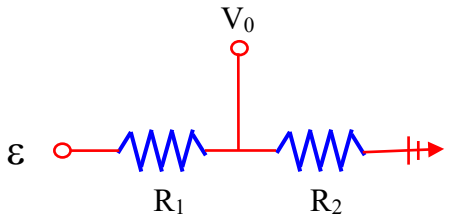



**Figure 7-10:** Voltage divider.

The current flowing through both series resistors is given by $I = \mathcal{E}/(R_1 + R_2)$. This current, following Ohm's law, will cause a voltage drop across the resistor $R_2$. This voltage drop with respect to ground is given as:

$$V_0 = I \times R_2 = \frac{R_2}{R_1 + R_2}\mathcal{E} = \frac{\mathcal{E}}{1 + R_1/R_2} \qquad [\ 7\text{-}7\ ]$$

By selecting a proper ratio of $R_1/R_2$, we can obtain any voltage $V_0$ between zero and $\mathcal{E}$.

Figure 7-10 is an electronics style drawing. Here we use the symbol "ground" to indicate common connection (or common node). Any voltage indicated without additional reference information is taken with respect to the ground.

In circuits we can talk about the voltage or potential *drop* across one or more elements of a circuit. This is what is measured by a voltmeter connected in parallel with the circuit elements. We can also talk about the voltage or potential of a point. The voltage at a point is simply the voltage of the point with respect to ground, that is, what a voltmeter would read if connected to that point and to ground. See Figure 7-11.

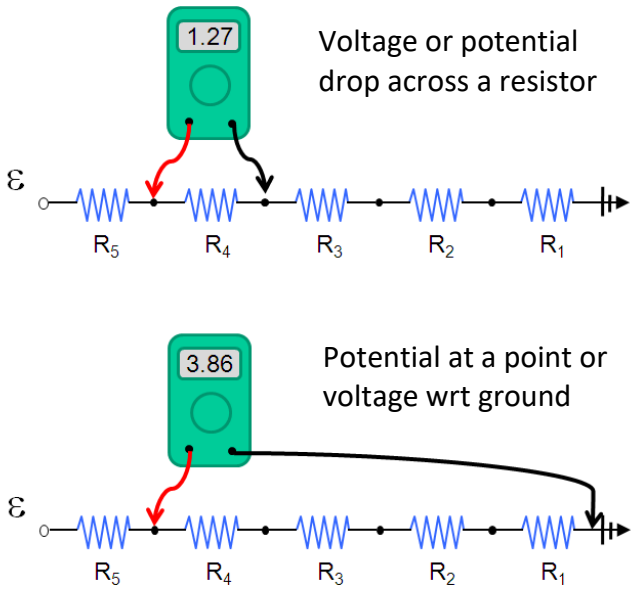



**Figure 7-11:** Voltage terminology

## Potentiometer

A voltage divider can be conveniently built using a variable resistor called a *potentiometer* (or *pot* for short). A potentiometer has the sliding electrode (a wiper), the position of which can be manually adjusted at a desired location resulting in a variable resistance. A voltage divider using a potentiometer is shown symbolically in Figure 7-12.

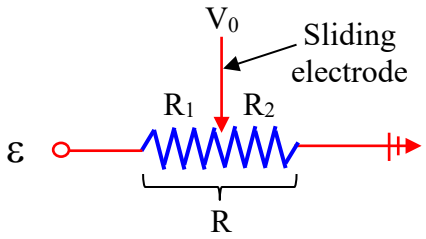
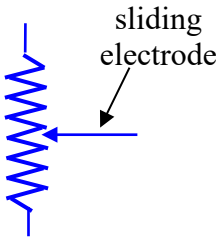



**Figure 7-12:** Potentiometer

The total resistance of the potentiometer is $R = R_1 + R_2$. A potentiometer usually has the form of a round drum with a central rotating shaft that controls the position of the wiper as shown in Figure 7-13. The 10 KΩ potentiometer we use in the lab is adjusted by a small screwdriver. You cannot tell just by looking at the potentiometer the resistance, $R_1$, $R_2$, or R, between any two leads. We recommend using an ohmmeter to check.




Schematic symbol

**Figure 7-13:** Potentiometer

## Diodes

A diode is an electrical "valve" that conducts current only in one direction – from Anode to Cathode**.** A special diode, called a Light Emitting Diode (LED), emits light whenever there is current flowing through it. An LED is often used as an indicator of the presence (LED is "on") or absence (LED is "off") of a voltage.

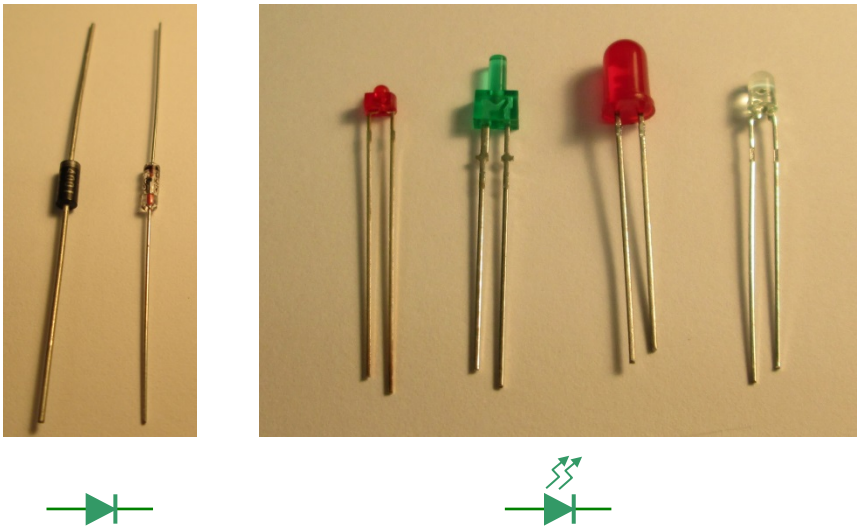


**Figure 7-14:** Diodes and LEDs

A simple circuit with LED is shown in Figure 7-15.

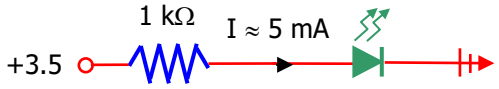



**Figure 7-15:** LED in action.

A small voltage $V_{forward}$, of between 0.5 and 2 V, must first be applied to a diode before a current can pass through it. Diodes are non-ohmic meaning that they do not obey Ohm's Law. To apply Kirchhoff's Rules to a circuit containing a diode we use a simple model. If the diode is forward biased, assume a constant $V_{forward}$ for the diode. This is a reasonable approximation if the resistances in the branch are not huge. If the diode is reverse-biased, the voltage drop over the diode is unknown but we can assume that the current in the branch with the diode will be zero. The actual current is not zero, just very, very small – on the order of tens of μA.

## Examples – KR and diode circuits

Consider the circuits below where the $V_{LED} = V_{forward} = 2$ V. Find the voltage drop across each circuit element and the current through each resistor.


*Solution*

In both cases (i) and (ii), applying Kirchhoff's Rules give the same equation

$$12 \text{ V} = V_{R1} + V_{LED} + V_{R2}$$

For (i) we assume that the LED is forward biased so that $V_{LED} = 2$ V. The resistors are in the same branch so must carry the same current. Resistors do obey Ohm's Law so we have

$$12 \text{ V} = I(3000\ \Omega) + 2 \text{ V} + I(2000\ \Omega) .$$

So I = 2 mA and $V_{R1} = 6$ V and $V_{R2} = 4$ V. The LED was forward biased since I is positive.

For (ii) the LED is reverse biased so that $V_{LED}$ is unknown. However the current in the branch is zero so by Ohm's Law the voltage drops over the resistors are zero,

$$12 \text{ V} = 0 + V_{LED} + 0 .$$

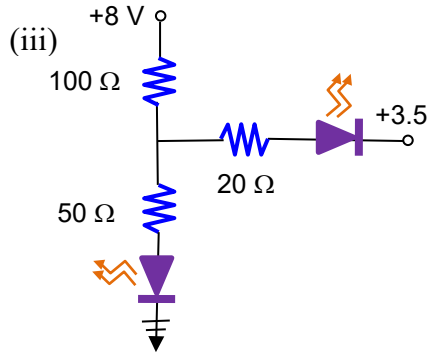
So the voltage drop over the reverse biased LED is 12 Volts.

For (ii) suppose we were not sure that the LED was reverse-biased. How do we check? Well we assume it is forward biased with a 2 volts drop from right to left and that a current I also flows right to left. The equations is then

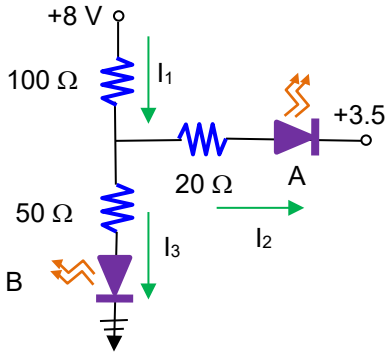
$$12 \text{ V} = -I(3000\ \Omega) - 2 \text{ V} - I(2000\ \Omega) .$$

Solving for I we find I = –14/5000 A. Since the current is negative, left to right, and it cannot flow that way through a diode, the current is actually zero and the LED is reverse biased.

For circuit (iii) below, determine if the LEDs light up. What is the current through each resistor? If an LED does not light up, what is the voltage drop across that LED? Take $V_{LED}$ = 2 V.




*Solution*
LEDs light up if they are forward biased. We don't know if the LEDs are forward biased, so we will assume that they are and that the currents flow in the correct direction for forward bias.




Our equations for the circuit are:

$$I_1 = I_2 + I_3$$
$$8\text{ V} = (100\ \Omega)I_1 + (50\ \Omega)I_3 + V_{LEDB}$$
$$5\text{ V} = -V_{LEDA} - (20\ \Omega)I_2 + (50\ \Omega)I_3 + V_{LEDB}$$

Solving the system of equations using $V_{LEDA} = V_{LEDB}$ = 2V, and not showing the details, we get $I_1$ = 17/800 A, $I_2$ = –9/160 A, and $I_3$ = 62/800 A. Since the direction of $I_2$ is impossible, the first LED A is actually reverse biased, and $V_{LEDA}$ is unknown and must be found. As well, we can assume $I_2$ = 0. Our equations become:

$$I_1 = 0 + I_3$$
$$8\text{ V} = (100\ \Omega)I_1 + (50\ \Omega)I_3 + (2\text{ V})$$

$$5\text{ V} = -V_{LEDA} - (20\ \Omega)(0) + (50\ \Omega)I_3 + (2\text{ V})$$

And we find $I_1 = I_3 = 6/150$ A, and $V_{LEDA} = -1$ V.

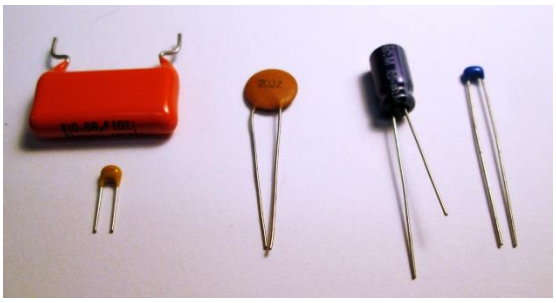
## Capacitors

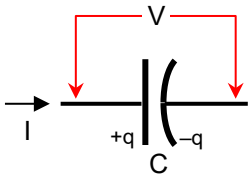
A capacitor, in its simplest form, is just a pair of parallel metal plates. When equal and opposite charges $+q$ and $-q$ are placed on the plates, there is a voltage difference that depends on the capacitance of the particular configuration

$$V = q/C. \qquad [\ 7\text{-}8\ ]$$

The symbol for a capacitor is shown below in Figure 7-16.

As you put charge on a capacitor, it gets harder and harder and takes longer and longer to add more charge. The diagram below shows a capacitor-charging circuit and a plot of the voltage across the capacitor plates as a function of time.






Schematic symbol

**Figure 7-16:** Capacitors and symbol.

The equation for a capacitor being charged from zero is given by the formula

$$V_C(t) = V\left(1 - e^{-\frac{t}{RC}}\right). \qquad [\ 7\text{-}9\ ]$$

The quantity RC is called the time constant τ of the circuit. When τ is small, the capacitor charges quickly; when it is large, the capacitor charges slowly. After 5τ, the capacitor is over 99% charged. A convenient way of thinking of a capacitor is that it acts as a switch. An uncharged capacitor is

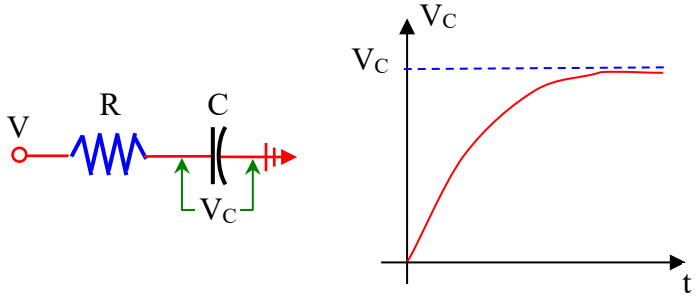



**Figure 7-17:** Charging a capacitor.

like a closed switch with current at its maximum. A charged capacitor is like an open switch, no current can flow.

When disconnected from a battery, a capacitor can hold its charge for minutes or even hours before it leaks away. Providing a path between the plates discharges the capacitor much more quickly. The discharging circuit and the $V_C$-t graph are shown in Figure 7-18 below.

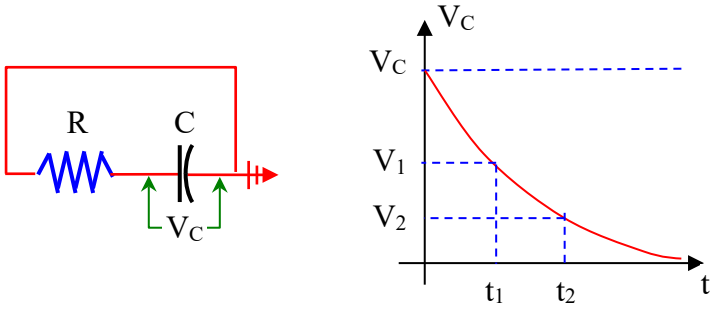



**Figure 7-18:** Discharging a capacitor.

The equation for a capacitor being charged from zero is given by the formula

$$V_C(t) = V_C e^{-\frac{t}{RC}}. \qquad [\,7\text{-}10\,]$$

When the time constant τ is small, the capacitor discharges quickly and when it is large, the capacitor discharges slowly. After 5τ, the capacitor is over 99% discharged. The discharge of a capacitor can be used for timing if you can measure voltage much faster than τ. Consider the two voltages $V_2$ and $V_1$ shown on the graph in Figure 7-18. Using Eqn [IV -10], the time difference between the voltages is

$$t_2 - t_1 = RC \ln\left(\frac{V_1}{V_2}\right). \qquad [\,7\text{-}11\,]$$

Since we are dealing with DC circuits, we may want to know initial currents and voltage drops in a circuit so that the PIC is safely buffered as well as the steady state or $t \rightarrow \infty$ case values. We can

readily apply Kirchhoff's Rules to the resistor-capacitor circuits for these two limiting cases. For the $t = 0$ case, note that the voltage drop across the capacitor is known. It will be zero as given in equation 7-12. This is equivalent to treating the capacitor as a *short*. For the $t = \infty$ case, the voltage drop across the capacitor is unknown but constant. However, a fully charged capacitor stops current from flowing in the branch where it resides, the capacitor acts like an *open*, so the voltage drop across any resistor in the same branch will be zero.

Capacitors also come in handy when you want to measure a varying voltage. An ordinary DC voltmeter reading the signal would fluctuate. A "sample and hold" circuit can be built using a very fast switch, a series resistor, and capacitor whose time constant $\tau = RC$ is much smaller than the time variation of the voltage signal. The switch is closed long enough for the capacitor to charge nearly completely, then opened. Since a voltmeter has a huge resistance, connecting it across the charged capacitor creates a circuit with a large time constant, thus allowing for a more leisurely measurement.

## Transducers

Transducers are electronic devices whose physical properties, usually voltage output or resistance, change with variations in its surroundings such as temperature, pressure, ambient light, and the like. For example, you may have learned that the resistance of a resistor depends on its temperature. However, this effect is weak when the temperature changes are only a few degrees and it would be impractical to use a resistor as a thermometer. A thermocouple, on the other hand, is very sensitive to temperature and is often used as a high temperature thermometer. A thermocouple utilizes the Seebeck Effect in which two dissimilar metals in contact with one another develop a potential difference. This potential difference is strongly dependent on temperature. In general, the relationship between the external effect one wishes to measure, and the physics property of the transducer, is nonlinear or, at best, only approximately linear over some range. As well, if the voltage output is small, the result may have to be amplified.

# Chapter 8. Digital Input Output (DIO)

Most of the pins on the MCU can handle DIO. Digital means having discrete, as opposed to continuous or analog, values. Here there are two discrete states 0 or 1 which can have several meanings: OFF or ON, FALSE or TRUE, LOW or HIGH voltage. For a pin, digital output means that the pin acts as a power supply that can be set to a high voltage (+.5 V) or to a low voltage (0 V or ground). Both these values are approximate, but no other values are possible.

Digital input means that the pin acts as a voltmeter that can detect whether the pin is connected to a high voltage (+5 V) or to a low voltage (0 V or ground). If you connect the pin to 6 V, it will probably read that voltage as high. If you connect the pin to 1 V, it will probably read that voltage as low. If you connect the pin to 2.5 V, it probably will not be able to decide if the input is high or low and it may randomly oscillate between reading high and low. Note the "probably". The datasheet specifies which ranges of voltages will be read as high and which as low.


**Figure 8-1:** Xpress board pin-out.

The pins on the MCU that have DIO functionality are grouped into sets of eight called ports. On the PIC18F46K42 we have five ports: PORTA, PORTB, PORTC, PORTD, and PORTE. The PORTA pins are labelled RA0 to RA7 on the MPLAB Xpress board shown in Figure 8-1 below. For example, RA0 is pin 2. Similarly the other port pins are labelled RBn, RCn, RDn , and REn where n is 0 to 7.

We configure the pins for DIO using the MPLAB Code Configurator (MCC) plug-in. In MPLAB X IDE, navigate to the Pin Manager Grid View as shown in Figure 8-2. In the two rows labelled Pin Module, you can select which pins are input and which are output. The blue unlocked symbol means that the pin is available to be selected. When you click on a blue unlocked square, it turns green with a locked symbol. Some pins are grayed out or orange meaning that they are already in use elsewhere.

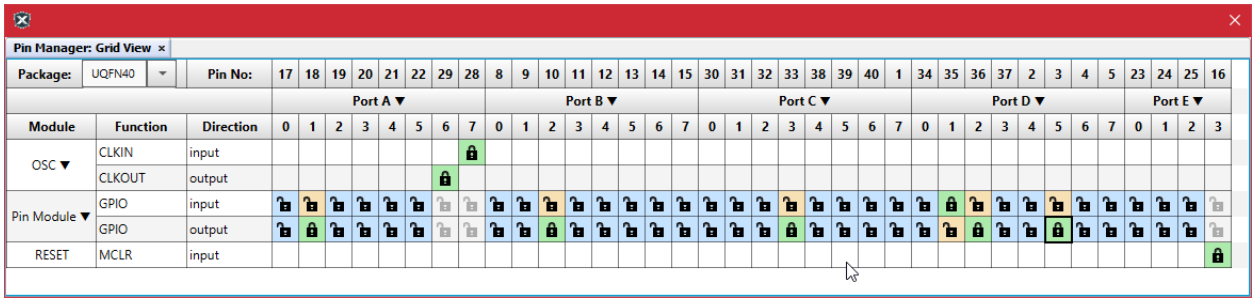



**Figure 8-2:** Pin Manager Grid. Unlocked blue pins and selected green locked pins.

DIO pins are, by definition, digital pins. Navigate to the Pin Module shown in Figure 8-3 and ensure that the Analog boxes are unchecked. In the Pin Module, you can give the also customize the default names IO_RXn to something more meaningful such as Button1_RA1 or GreenLED_RB2.

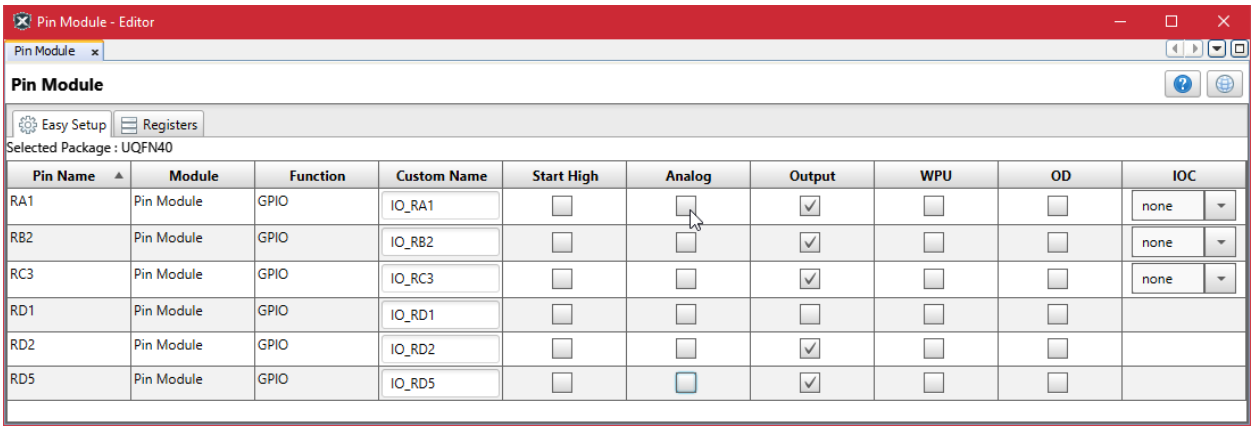



**Figure 8-3:** Pin Module. DIO pins must not be set to Analog.

When you generate the code in MCC, the pin_manager.h and pin_manager.c library files will be created which will contains functions for using the selected pins. The most useful functions are given in Table 8-1. If you had customized the name of a pin to GreenLED_RB2, the functions would be named `GreenLED_RB2_SetHigh()`, `GreenLED_RB2_SetLow()`, etc.

**Table 8-1:** DIO functions for generic pin names.

| Function | Description |
|---|---|
| IO_RXn_SetHigh() | Set digital output pin RXn* to high. |
| IO_RXn_SetLow() | Set digital output pin RXn to low. |
| IO_RXn_Toggle() | If digit output pin RXn is high, change to low and vice versa. |
| IO_RXn_GetValue() | Read whether digital input pin is high Returns 1) or low (0). |

*X = A, B, C, D, or E. n = 0 to 7.

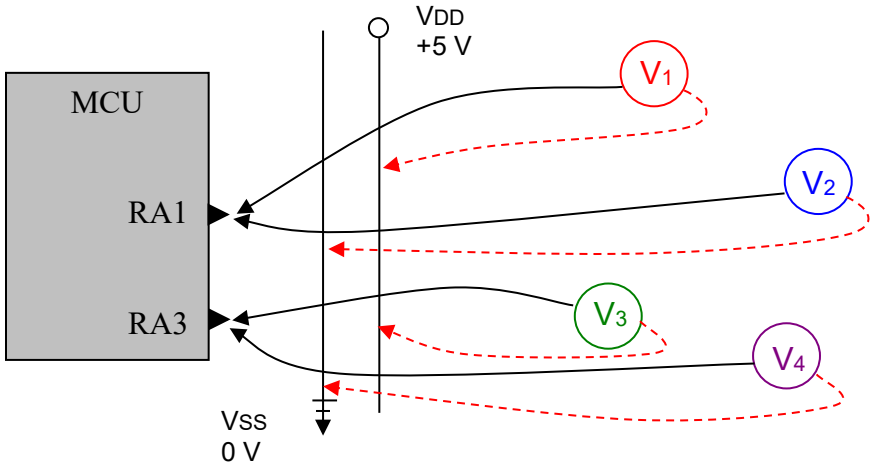
If you had customized the name of a pin to GreenLED_RB2, the functions would be named `GreenLED_RB2_SetHigh()`, `GreenLED_RB2_SetLow()`, etc.

Let's use this last bit of code to write a program that would actually set these pins when loaded onto the PIC.

```
//  testDIO.c Use MCC to configure and set RA3 high and RA1 low
#include "mcc_generated_files/mcc.h"  // library for SYSTEM_Initialize()

int main(void)
{
  // Initialize the device - MCC file configures clock and DIO
  SYSTEM_Initialize();
```

```
   IO_RA3_SetHigh();
   IO_RA1_SetLow();

   while(1);  // run forever
}
// end program
```

Q. The diagram below represents your MCU programmed with the previous code. The vertical lines represent the power and ground lines of the breadboard. Four voltmeters are connected to the breadboard and pins on the MCU. Note that when a red (dashed) lead (V on the DMM) is connected to a higher voltage point and a black (solid) lead (COM on the DMM) is connected to a lower voltage point, the voltmeter will register a positive value. Reverse the connections and the value will be negative. What is the reading on each voltmeter?




$V_1$ = _____ $V_2$ = _____

$V_3$ = _____ $V_4$ = _____

Q. How would you change the code example to change the sign of the sign of the voltage readings?

## Digital Output and Controls

Digital output can be used to control devices attached to the MCU. This can be as simple as turning on a light when certain conditions are met to operating more complex devices such as a liquid crystal display (LCD).

The key to control is using the pins to provide power to circuits. Figure 8-4 shows the simplest circuit for each pin – a single resistor.

The diagram shows the diagram of the conventional current (i.e. running from high to low).
The current is determined from Ohm's law, I = V/R, where V is the voltage difference across the resistor.

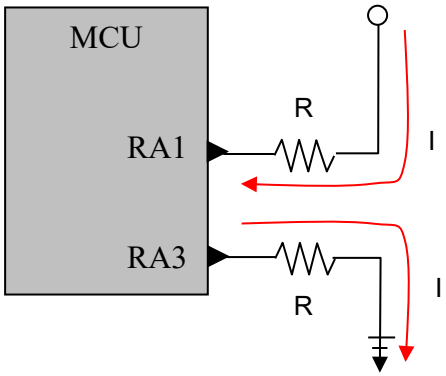



**Figure 8-4:** Powering a circuit using a pin.

WARNING! The pins and MCU have a maximum current they can provide (source) or that they can accept (sink). Exceeding these limits can alter the operation of the MCU or burn it out. According to Section 26.0 of the PIC18F46K42 datasheet, the maximum current sourced or sunk by any pin is 50 mA. The total current that can be sourced or sunk by all the pins operating at the same time is 200 mA.

The pin can supply 5 V and a maximum current of 50 mA. From this we can determine the smallest equivalent resistance that any circuit attached to a pin can have, R = 5 V / 50 mA = 100 Ω. Any resistor of 100 Ω or bigger should be safe to use with the MCU.

Since we cannot see currents, the circuit in Figure 8-4 is not very impressive. To see something happen, we usually add LEDs to the circuit as shown below in Figure 8-5. LEDs are convenient to use because they require less than 5 volts to operate. LEDs have a very small resistance of around 10 Ω, so we must use them in series with a resistor to keep the pin currents within acceptable limits. This also protects the LEDs themselves. LEDs will burn out if the current is too great. Connecting an LED across 5 V without an added resistor is likely to burn it out. LEDs are diodes and diodes only carry current in one direction. Therefore, how you connect the LED to a pin is important. Here are two LED circuits that will light up an LED using the previous code `testDIO.c`.

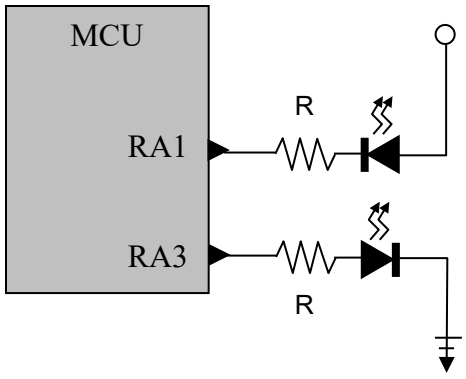



**Figure 8-5:** Controlling LEDs.

## Digital Input and Sensors

Digital input can be used to let the MCU sense the outside world. This can be as simple as determining whether a button is pressed or as complex as interpreting a digital signal from another device such as a printer. The key is that the signal be encoded as high and low voltages.

It monitors two input pins, and depending on their status, lights up two LEDs attached to two output pins (see circuit schematic Figure 8-6 after code).

```
//  LEDinputsensor.c Use MMC RA3 and RA2 as output (LEDs) and RA1 and RA0
//                 as input (buttons)
#include "mcc_generated_files/mcc.h"  // library for SYSTEM_Initialize()

int main(void)
{
 // Initialize the device – MCC file configures clock and DIO
 SYSTEM_Initialize();


 while(1)  // monitor and read pins and light LEDs forever
   {
    // whatever the value of input at RA1 use that value to turn RA3 on/off
    if( IO_RA1_GetValue())
      IO_RA3_SetHigh();
    else
      IO_RA3_SetLow();

   // whatever the value of input at RA0 use that value to turn RA2 on/off
   if( IO_RA0_GetValue())
      IO_RA2_SetHigh();
    else
      IO_RA2_SetLow();
   }
}
// end program
```

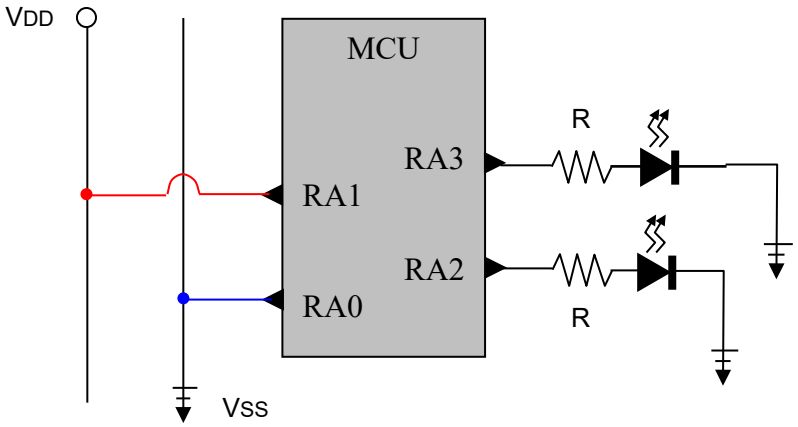



**Figure 8-6:** Inputs and outputs.

In the circuit in Figure 8-6 above, the LED at RA3 would be lit and the LED at RA2 would be off. The nice thing about this circuit is that you can change the connections at RA1 and RA0 as you wish.

Q. How would you change `LEDinputsensor.c` so that if RA1 is high, RA3 would be set low (LED off) and the same for RA0 and RA2?

Q. If you do not change `LEDinputsensor.c` above, how could you change the LED circuits at RA3 and RA2 so that if RA1 is high then RA3 is set low (LED off) and the same?

## Buttons and DIO

Changing the voltage at an input between low and high when you wish is easily accomplished by a simple circuit of two resistors and a button as shown in Figure 8-7.

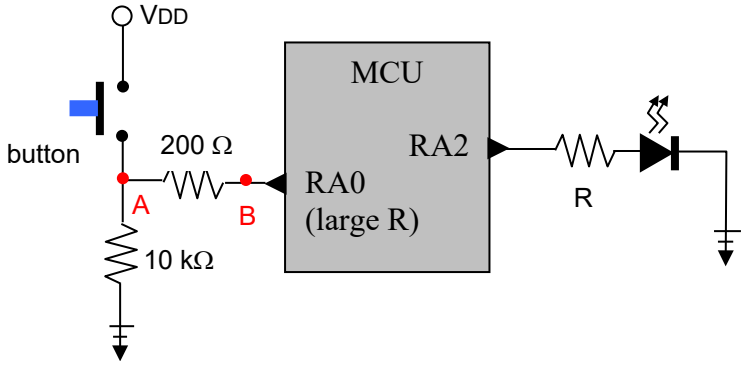



**Figure 8-7:** A button input.

We need to spend a little time understanding how the button works. When the button is up, RA0 is connected to ground through the two resistors, so the voltage at the pin is 0.

The button is just a conducting piece of metal so when it is pressed down, point A is at the same potential as VDD, +5 V. What will be the voltage at RA0, point B, in this case? We can figure this out if we remember that the input pin has a large resistance on the order of millions of Ohms. We can model the button circuit and the MCU as shown in Figure 8-8 below.

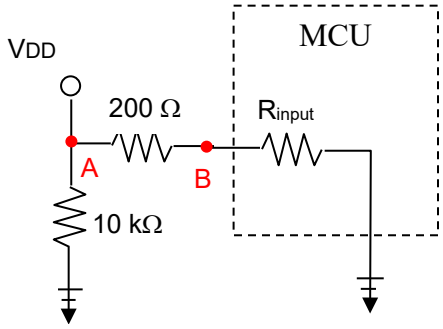



**Figure 8-8:** Input resistance.

Point A is at 5 Volts when the button is pressed and there are two parallel paths to ground. Because both paths are high resistance, very little current is flowing. That is good because it would just be wasting power. The branch to the input pin is a series circuit of the 200-Ω resistor and $R_{input}$. If we recall the voltage divider circuit, the voltage at point B is easy to find

$$V_B = V_A * R_{input} / (200\ \Omega + R_{input}).$$

Since $R_{input} >> 200\ \Omega$, $V_B = V_A$ to a very good approximation.

The voltage drop across the 200-Ω resistor would be negligible. The only purpose of the 200-Ω resistor in this circuit is to limit current if the wrong program is downloaded into the PIC and the I/O pin is configured as an output.

The correct way to construct the button circuit on your breadboard is shown in Figure 8-9 below. The 10 kΩ resistor connects one button pin to ground. The 200 Ω resistor connects the other pin to power. Check the resistance of both resistors with your DMM. The signal pin on the button is the one connected to the 10 kΩ resistor. Use your DMM to check the voltage at that pin. It should read 5 V when the button is pressed and 0 when open.

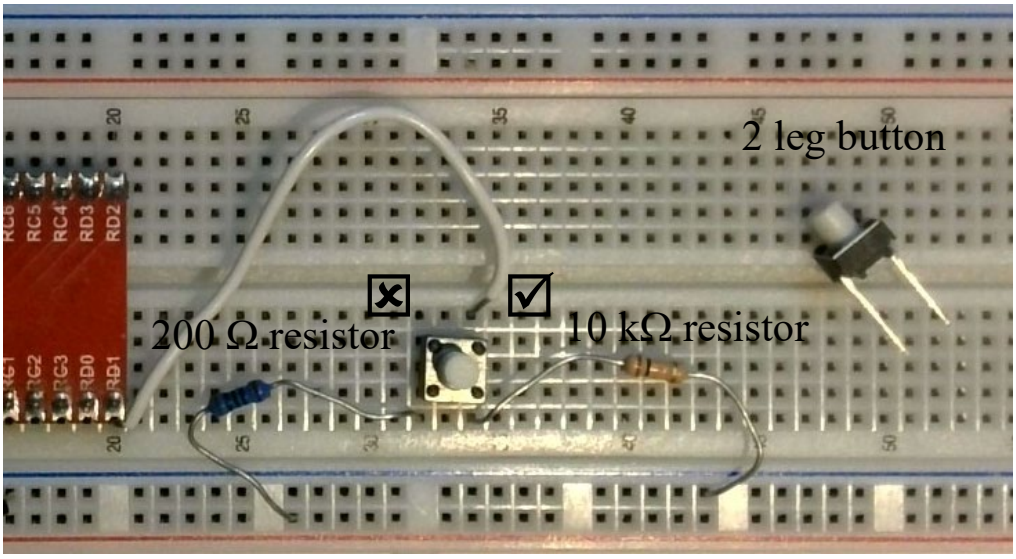



**Figure 8-9** Button circuit.

You may have been provided with several buttons which already have the 10 KΩ and 200 Ω resistors wired into a piece of circuit board. The two pairs of pins give you a firm connection to the power and ground rails of your breadboard. You just need to put a wire between the signal pin and the PIC MCU DIO pin. The buttons are wired for insertion on the left-hand side of the breadboard.

| Inserting the button on the right-hand side of the breadboard will cause problems! |
|---|

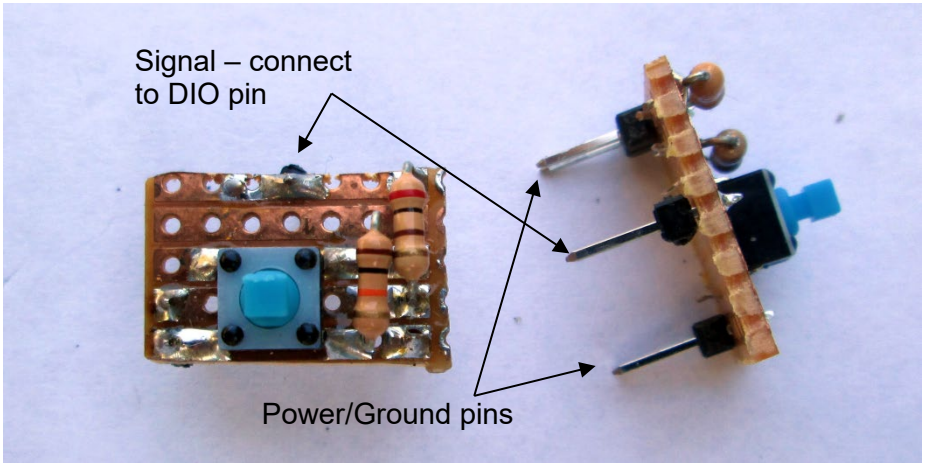



**Figure 8-10:** Prewired buttons

Now consider the program below which is written for a button.

```
// buttonLED.c Use MMC. RA2 as output. RA0 as input
#include "mcc_generated_files/mcc.h"  // library for SYSTEM_Initialize()

int main(void)
{
 // Initialize the device – MCC file configures clock and DIO
 SYSTEM_Initialize();

 while(1)  // monitor and read pins and light LED forever
   {
     // whatever the value of input at RA0 use that value to turn RA2 on/off
    if( IO_RA0_GetValue())
      IO_RA2_SetHigh();
    else
      IO_RA2_SetLow();
   }
}
// end program
```

With `buttonLED.c`, if the button is pressed on and off, the LED flashes on and off in unison.

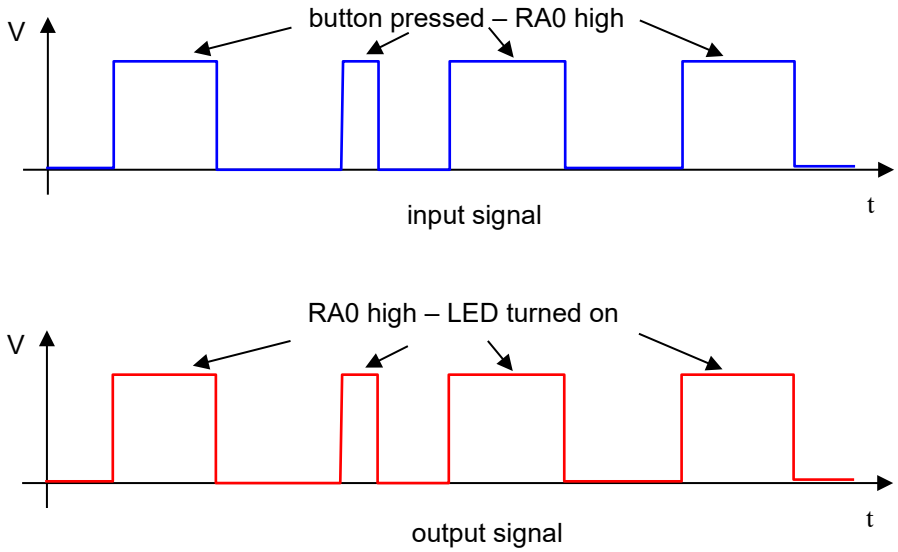



**Figure 8-11:** Button input and LED output.

Note that the button and resistor circuit is a logic switch. Modern digital logic devices require a discrete voltage level to work reliably. In the case of the PIC, the input will be taken as a high if the input voltage is above 2 volts DC and the input will be taken as a low if the input is below 0.8 volts DC. If the voltage is between 0.8 and 2.0 volts DC, we can't predict what the input will be. This circuit set the input pin voltage high or low but with very little current flow. This is in contrast to a wall light switch in a house circuit which allows a large current to flow through a lightbulb filament, a resistor, to make it glow (see Figure 8-12 below).

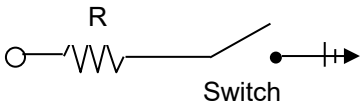
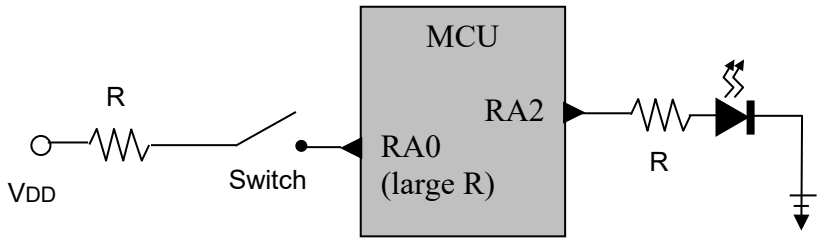



**Figure 8-12:** A mechanical switch.

Q. Why don't we want to use a mechanical switch in series with the MCU as shown in the diagram below?



You can also use the button and resistor circuit as a toggle by modifying how the program reacts to the change of state, either low to high (a rise) or high to low (a fall), at the input. With a toggle, when you press and release the button for the first time, the LED should stay on. When you press and the button the next time, the LED should remain off. This behaviour is illustrated below.

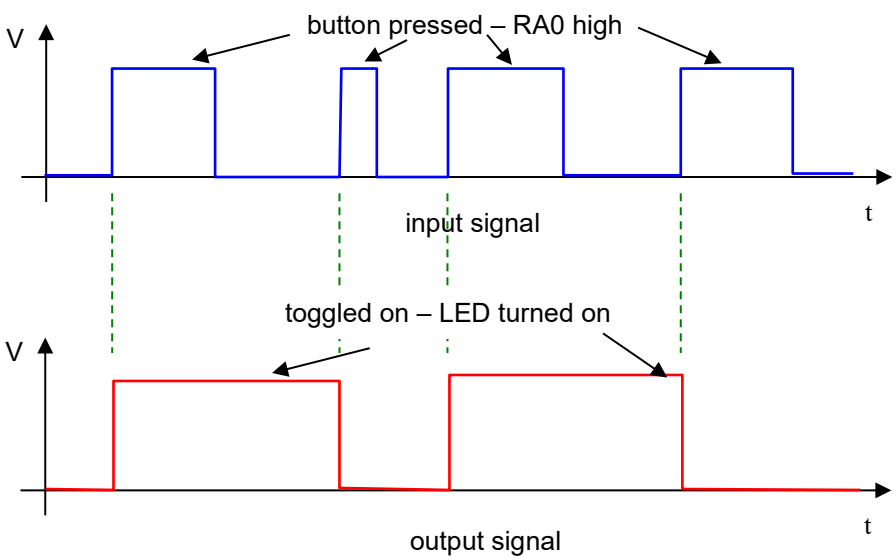



**Figure 8-13:** Toggle on rise logic.

To create a toggle in software, your code must look for the times that the input signal goes from low to high (known as a *rising edge*) or high to low (*falling edge*). Source code to do this is in button.c below.

```
// monitor a digital input pin RXn for changes
// We usually connect a button to this pin but any square wave input will do.

#include "buttons.h"

unsigned char last_switch1_value = 0;     // last button value, initialized to low
                                          // global variable

// returns 0 if no edge has happened
// returns 1 at a rising edge
// returns 2 at a falling edge

unsigned char monitor_switch1_for_edges(unsigned char digitalinputpin)
{

    unsigned char has_switch1_changed = 0;  // 0 = no change; 1 = rising edge;
                                            //  2 = falling edge

    if (last_switch1_value == 0 && digitalinputpin)
    {                // rising edge detected if digitalinputpin is 1 (on)
        last_switch1_value = 1;   // save current switch value (on)
        has_switch1_changed = 1;  // found a rising edge
    }
    if (last_switch1_edge == 1 && !digitalinputpin)
    {                // falling edge detected if digitalinputpin is 0 (off)
        last_switch1_value = 0;   // save current switch value (off)
        has_switch1_changed = 2;  // found a falling edge
    }

    return has_switch1_changed;
}
```

Note the elements of this program. There is the `monitor_switch1_for_edges()` function which sits inside the while(1) loop waiting to detect a rising or falling voltage signal or edge. It does nothing until the edge is detected. A key part of the function is the global variable `last_switch1_edge`. It holds the last reading from the input pin. That value is compared to the current value. If the previous value is 0 and the current value is 1, we have found a rising edge. If the values had been 1 and 0, we found a falling edge. If you need a second button, simply repeat the code replacing the number 1 by 2.

This function is then put to work in a program like

```
// SoftToggle.c
// button_RD1 to act like a switch or toggle for LED on RD2
// MCC digital input button_RD1, digital output led_RD2

#include "mcc_generated_files/mcc.h"
#include "buttons.h"

void switch1_risingedge_action(void);
void switch1_fallingedge_action(void);

int main(void)
{
    // Initialize the device
    SYSTEM_Initialize();

    unsigned int has_switch1_changed = 0;

    while(1)
    {
      has_switch1_changed = poll_switch1_for_edges(button_RD1_GetValue());
      if ( has_switch1_changed == 1 ) switch1_risingedge_action();
      if ( has_switch1_changed == 2 ) switch1_fallingedge_action();
    }
    return 0;
}


void switch1_risingedge_action(void)
{
   led_RD2_Toggle();      // flip LED status
}

void switch1_fallingedge_action(void)
{
   // not used in this example
}

// end program
```

The two functions `switch1_risingedge_action()` and `switch1_fallingedge_action()` are where you add your code to make the program do what you wish it to do.

Q. How would you modify the code to change on a fall (see Figure 8-14)?

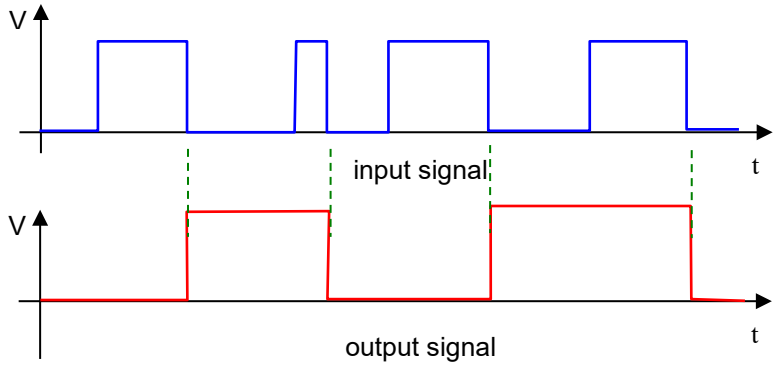



**Figure 8-14:** Toggle on fall.

# Chapter 9. Processor Speed & Delays

The MCU has an internal clock or oscillator that synchronizes the operation of its various modules. The default speed of this internal clock is 1 MHz but we can change that if we wish. The internal clock output is a square wave, shown in Figure 9-1, at this frequency. The manual indicates the system clock frequency and period by FOSC and TOSC, where FOSC = 1/TOSC.

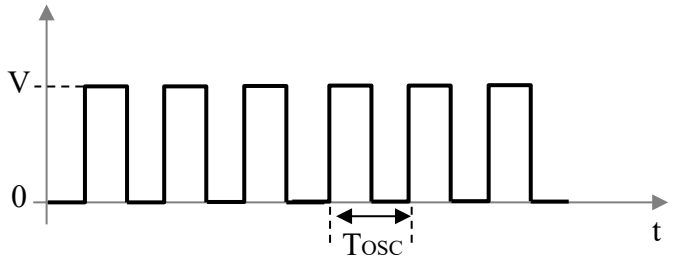



**Figure 9-1:** Internal oscillator square wave.

Every operation, storing a number in memory, adding two numbers, or setting a port pin requires time. The minimum amount of time required on this chip, because of the limits of the computer architecture, is 4TOSC. This minimum time is called an instruction cycle TCY = 4TOSC.

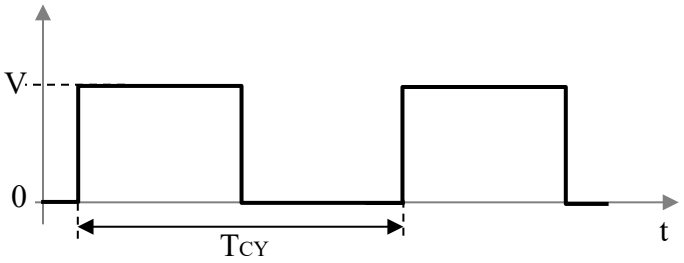



**Figure 9-2:** The instruction cycle signal.

The MPLAB has a method of letting us find out how many instruction cycles, and how much real time, a C statement would take to execute by simulating the running of the program. If you set "breakpoints" in the code, you can stop execution at those points. With the stopwatch feature selected, you can time from one breakpoint to another.

Figure 9-3 show the breakpoints (the red squares beside the line numbers) and stopwatch window for a sample program. The green arrow indicates the current line that the simulator is at, i.e. line 2. The stopwatch window indicates that 1TCY, or 1 μs at 1 MHz, were required to process the calculations between this line and the previous breakpoint.

The program has comments showing how many instruction cycles are needed for each line.

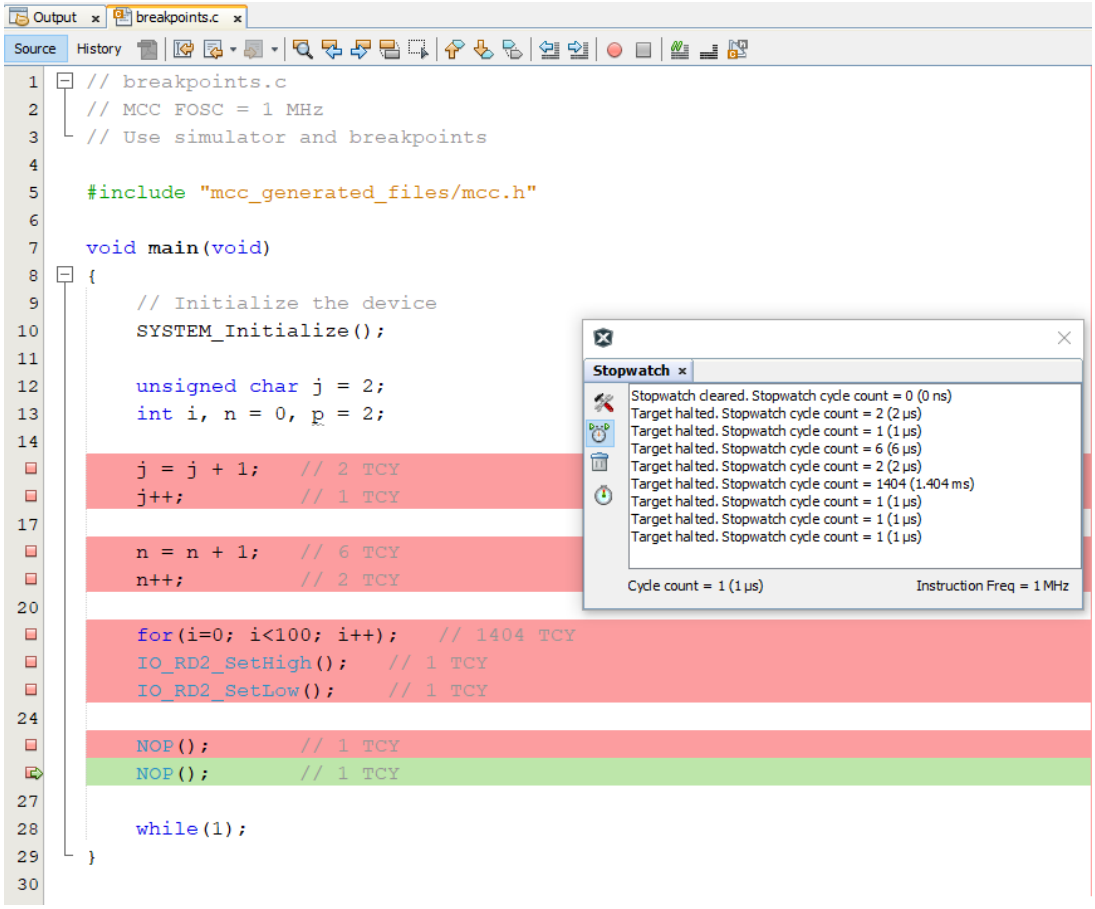



**Figure 9-3:** Timing code interval in the simulator.

Note that j = j + 1 (line 15) takes less time than n = n + 1 (line 18) because j is a smaller sized variable than n and thus takes less computing time since there is less memory to juggle.

See also j++ (line 16) is optimized and thus takes less time than j = j + 1 (line 15). The same is true for n = n + 1 versus n++ on lines 18 and 19.

For loops take a lot of instruction cycles to process.

A breakpoint can only be inserted on a line of code and not at a blank line. If there is no code available where you want the breakpoint, you can insert a NOP() function and place the breakpoint on the NOP(). Think of NOP() as a do-nothing function. It takes exactly 1 instruction cycle to use a NOP().

| Q. How long is 1 TCY on an MCU operating at 1 MHz? At 16 MHz? |
|---|

## Changing the MCU Clock Frequency

The PIC/MCU can operate at different speeds. The MPLAB Code Configurator (MCC) plug-in simplifies the process. In the MCC Resource Manager, open the System Module to make changes to the operating frequency. In Easy Setup, the top line tells you the *Current System clock* frequency commonly called FOSC. As you make changes, this line will also change. There are three choices to make. First in *Oscillator Select*, choose HFINTOSC (Figure 9-4 a). This stands for the High Frequency Internal Oscillator. The HFINTOSC can run at different speeds from 1 MHz to 64 MHz. Your second choice is to pick which frequency from the drop-down menu under *HF Internal clock* (Figure 9-4 b). 32 MHz is a safe choice. The last choice is *Clock Divider* which has a drop-down menu which goes from 1 to 512 in powers of 2 (Figure 9-4 c). Choose 4 and the operating frequency becomes FOSC = 32 MHz / 4 = 8 MHz. Clearly you can make the PIC/MCU run from FOSC = 64 MHz / 1 = 64 MHz down to FOSC = 1 MHz / 512 = 1.95 kHz.

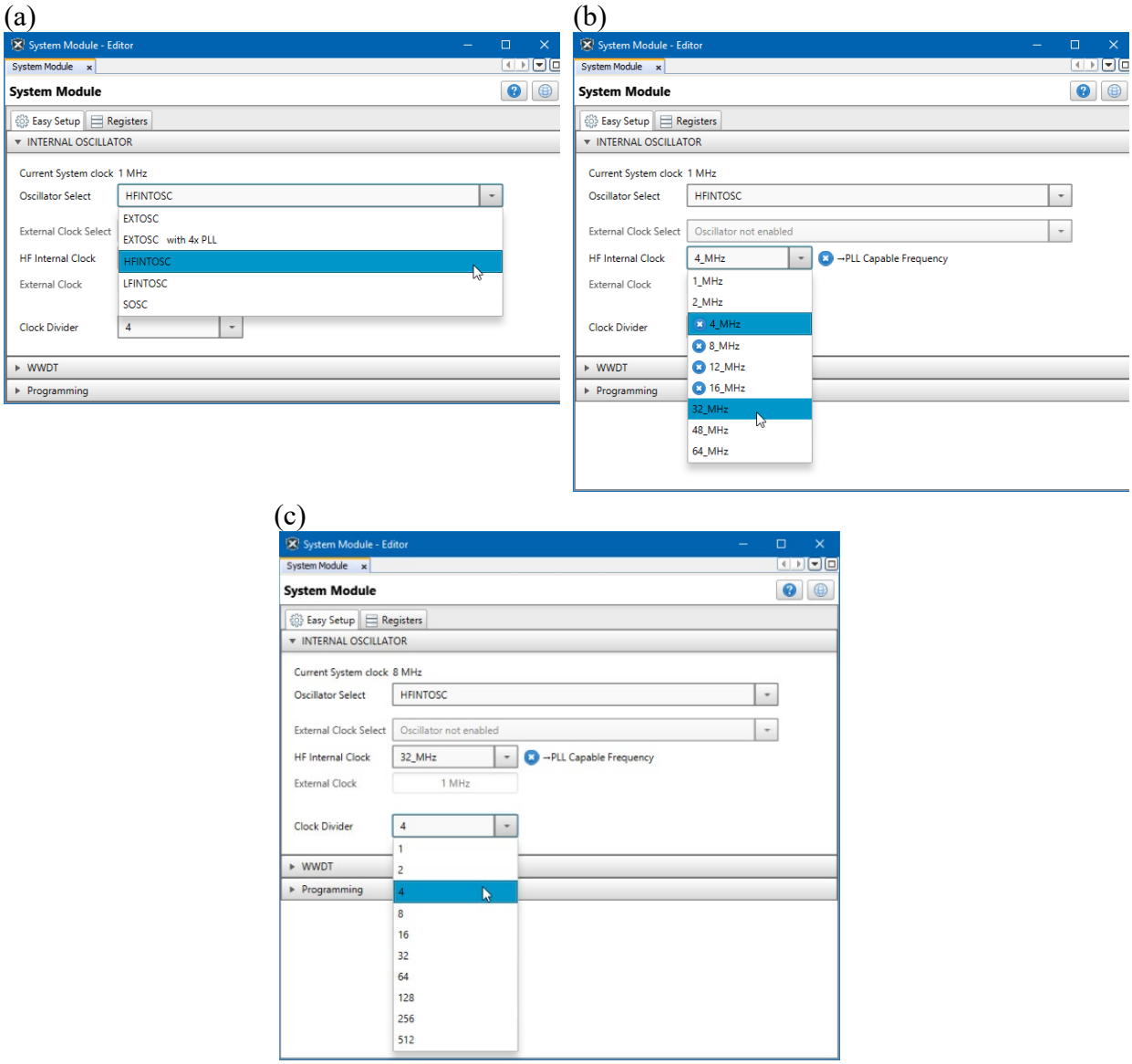



**Figure 9-4:** Setting FOSC in System Module

If you wish to use the value FOSC in calculation in your code, the variable `_XTAL_FREQ` holds the current value.

## Determining the Clock Frequency of the MCU

Pin RA6 can be used to measure the FOSC and TOSC. In the IDE tab *Pin Manager Grid View*, select RA6 as shown in Figure 9-5. When the code is generated, built, installed on the PIC/MCU, pin RA6 will output a square wave with a period TCY, of period FOSC/4. Measure the signal with an oscilloscope or the frequency setting on the DMM.

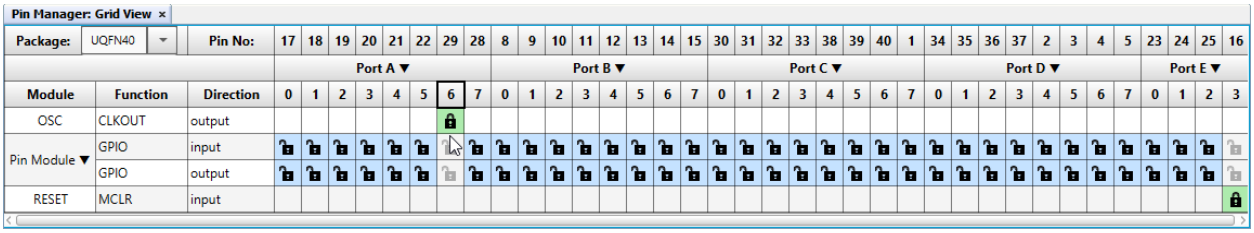



**Figure 9-5:** CLKOUT on RA6

## Using am External Oscillator

The internal oscillator of the PIC18F46K42 is only accurate to a few percent. So, for example, if you generate a PWM signal at 1000 Hz, it may only be 1000 Hz ± 2.3%. It is possible to have oscillators of much higher accuracy. Such precise oscillators are in just about every timepiece. These seldom lose more than a second over a day. There is a tradeoff, with the internal oscillator you can switch among a wide range of frequencies from 1 MHz to 64 MHz. The external oscillator is fixed at one frequency.

You can buy external oscillators at many different frequencies. Provided the frequency of the external oscillator is not outside the specifications of the PICMCU, any of them can be used. We have provided you with such an external oscillator.

To use an external oscillator, you need to configure the PIC18F46K42 appropriately and you need to build the appropriate circuit. Configuration is done in the *System Module* of the MCC *Resource Manager* Figure 9-6. In *Oscillator Select* choose *EXTOSC with 4x PLL*. PLL is circuitry that turbocharges an oscillator to four times its stated value. This means that an 8 MHz external clock is effectively 8 × 4 = 32 Mhz. Under *External Clock Select* choose *EC (external clock) for 500 kHz to 8 MHz; PFM to medium power*. Under *External Clock*, enter the frequency of the oscillator you have been given. This should be 8 MHz. *Clock divider* can be set to any value you like.

When you select an external oscillator above, the Pin Manager: Grid View changes to show that there is now a CLKIN (clock in) input pin as in Figure 9-7. It will be pin RA7. You will connect the output of your external oscillator circuit to this pin.

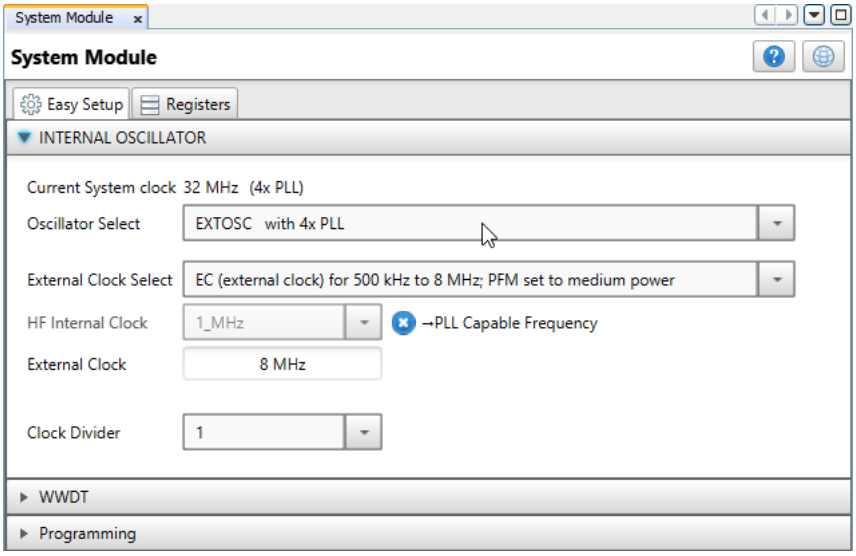



**Figure 9-6:** External oscillator configuration

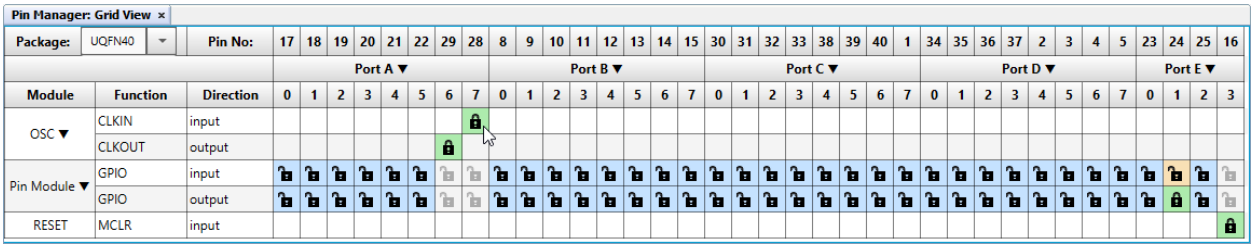



**Figure 9-7:** CLKIN pin for external oscillator

The external oscillator package has three pins. The package actually has four legs but the extra leg in nonfunctional, although it helps make the package fit snugly on a breadboard. The three active pins are ground, power, and signal output. The package has a small pointy corner labelled on top with a o to that you get the orientation correct, see Figure 9-9 . The active pins are not labelled so you must be careful. The package is inserted across the gutter on the breadboard, see Figure 9-8. A small capacitor is connected between the power leg and ground to suppress noise. Wires should be kept as flat and short as possible.

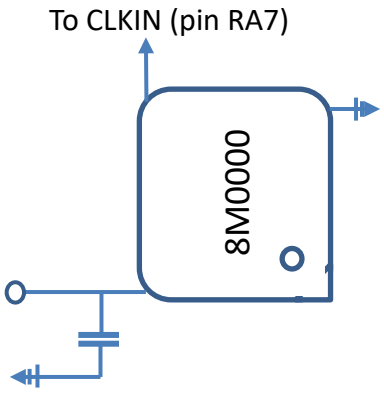



**Figure 9-9.** External oscillator circuit diagram.

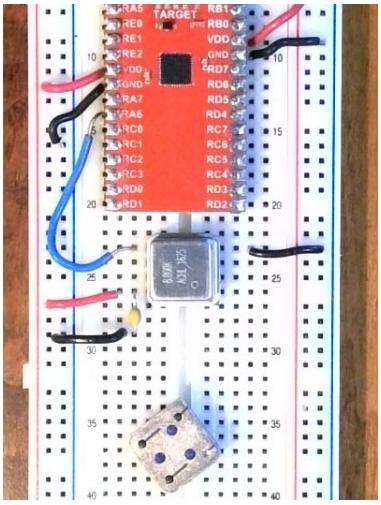


**Figure 9-8:** External oscillator on breadboard.

## Delays

In programming, it is useful to have a wait or delay function. For instance, you might want to blink an LED using logic like: LED on, wait 250 ms, LED off, wait 250 ms, repeat. MCC has a library of useful delay functions. In the MCC Resource Manager, under Device Resources and Foundation Services there is a DELAY module. See Figure 9-10. Add it to your project and you will have access to two functions `DELAY_milliseconds(n)` and `DELAY_microseconds(n)` to create your delays. Note that you cannot have a delay of less than 40 microseconds.

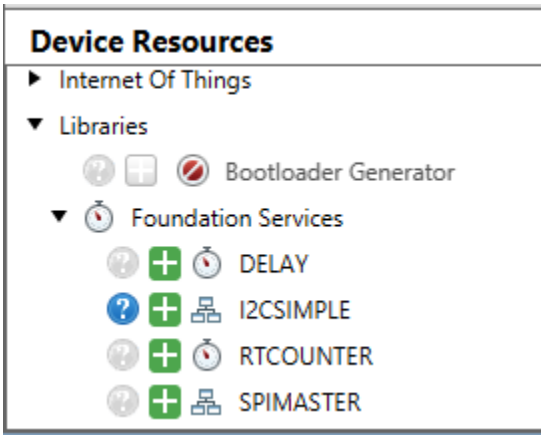



**Figure 9-10: DELAY library**

## DIO and Square Wave Output

It is easy to output a square wave from a DIO pin. The steps are simple:

- Turn pin on
- Wait a while
- Turn pin off
- Wait a while
- Repeat

```
// BlinkLED.c
// Simple Square Wave output, blinking LED
// MCC FOSC = 32 MHz, RD2 digital output, DELAY

#include "mcc_generated_files/mcc.h"

int main(void)
{
     // Initialize the device
     SYSTEM_Initialize();

     while(1)              // anything in here repeats forever
        {
         IO_RD2_SetHigh();       // set RD2 high, LED on
         DELAY_milliseconds(100) // wait 100 ms
```

```
            IO_RD2_SetLow();         // set RD2 low, LED off
            DELAY_milliseconds(80)   // wait 80 ms
           }
}
```

We could measure this signal using an oscilloscope connected to RD2 and ground, or we could also use the stopwatch feature to measure the on/off times in the simulator.

We define the *duty cycle* of a square wave as the ratio of the on time to the period of the square wave times 100% (duty cycle = $T_{on}/T \times 100$ %). Sometimes $T_{ON}$ is refereed to as the duty cycle or the pulse width.

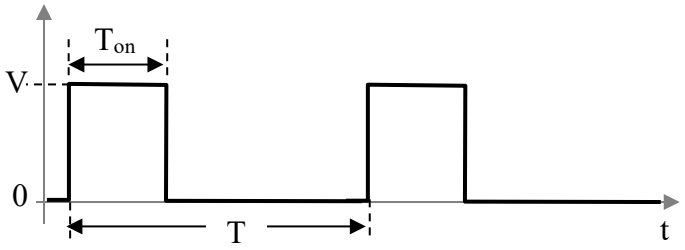



**Figure 9-11:** The duty cycle.

Q. What is the duty cycle for the above code?

Q. How would the output square wave differ if you connect the oscilloscope between RD2 and $V_{DD}$? What would the duty cycle be?

Q. What delays would you need for 0.250 ms period and a 32% duty cycle?

## Timing Diagrams

When the signal at one or more pins changes with time, the route to understanding the circuit behaviour is to draw the Voltage versus Time graph of the signals. For example, consider an LED and series buffer resistor connected between pins RA3 and RA5 on the MCU. When RA3 is high and RA5 is low, the LED will shine. The code controlling the pins is

```
//Square wave output on RA3 and RA5                                  //
// MCC FOSC = 32 MHz, RA3 and RA5 as digital output, DELAY

#include "mcc_generated_files/mcc.h"

int main(void)
{
     // Initialize the device
     SYSTEM_Initialize();

     IO_RA3_SetLow();      // initialize to low
     IO_RA5_SetLow();;     // initialize to low
```

```
    while(1)                // anything in here repeats forever
        {
         IO_RA3_SetHigh();
         DELAY_milliseconds(100);
         IO_RA5_SetHigh();
         DELAY_milliseconds(100);
         IO_RA3_SetLow();
         DELAY_milliseconds(200);
         IO_RA5_SetLow();
         DELAY_milliseconds(200);
        }
}
```

We draw the graph in Figure 9-12 to determine the period and duty cycle of the LED.

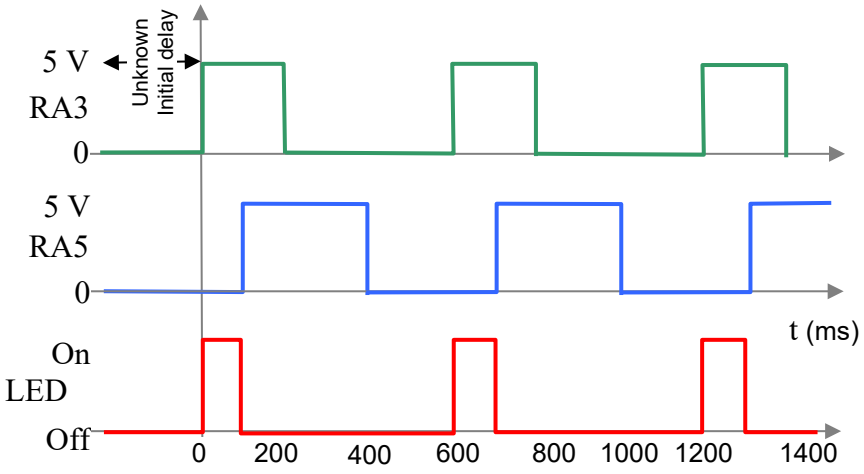



**Figure 9-12:** Timing problem solution via a timing diagram.

Reading Figure 9-12, the period is clearly 600 ms and the duty cycle is 100 ms.

# Chapter 10. Timers

Using either for-loops or delay functions means that nothing else can be done while we are waiting for the count to be finished. Such delays are called “blocking” delays. If we wanted to check a button, no luck. The programs are largely useless and not interactive. We can only do one thing at a time. Fortunately, the PIC18F46K42 designers and their predecessors in chip design were a clever bunch. These chips have separate modules that can run simultaneously. This chip has seven different timer modules. The simplest function of a timer module is like an egg timer. You set it for a certain amount of time, you start the timer, and you wander away and do something else. You can either check the timer periodically or you can listen for an alarm (if you have the fancier model).

A timer module works by checking the internal oscillator signals. It can tell time by counting either rising edges or at falling edges of the clock square wave signal. When you tell the module to start at a rising edge, it waits for that next rising edge and increments a counter for each successive edge. If the setup is done appropriately, when the desired number of edges is reached, a flag is set, see Figure 10-1. A flag is just a bit in an SFR. Setting the flag is the equivalent to the alarm sound on the egg timer. At any time, you can check this flag bit.

A counter, again an SFR, keeps a running tally of the number of edges counted. The counter can be byte-sized, 0 - 255, or two-byte sized, 0 - 65535. You can start (write) the counter at any value in its range. You can also read the counter at any time. Even 65535 edges, i.e. 65535 $T_{CY}$, is not a long time on a PIC run at 32 MHz. For that reason, you can have the counter increment only after multiple edges are counted such as every 2 successive edges, every 4 edges, and every 8 edges. This multiple is called the *prescale* value. Each timer module is slightly different in terms of which multiples or prescalars are allowed.

The counter, being limited in memory space, will roll over at maximum value called the timer *period*. For a 1-byte counter, the maximum value of the period is 255 although you may use a smaller period if desired. For a 2-byte counter, the maximum value of the period is 65535. When the counter is reached, it rolls back to zero and sets the timer interrupt flag bit to 1. Note: once the flag is set, it remains set (1) until it is cleared (0) manually. Figure 10-1 shows a timer being called at the red arrow. In the diagram, the timer starts at the next rising edge and the left blue arrow. The flag is set when the timer reached the period of 4TCY shown the right blue arrow.

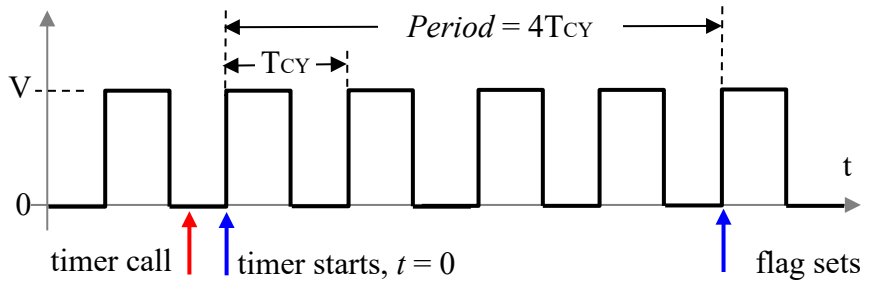



**Figure 10-1:** Timers start at the next edge (falling or rising as requested).

Some timer modules also have a *postscaler*. A postscaler determines how many times the counter must roll over before the flag is set. If a postscaler equals 2, the flag would not set until the counter rolled of twice or at $t$ = 8 TCY in the example above. If the postscaler was M, the flag would not be set until the counter had rolled over M times.

The seven timers are TMR0, TMR1, TMR3, TMR5, TMR2, TMR4, and TMR6. TMR1, TMR3, and TMR5 are 16-bit timers and are identical to one another except for the names. TMR2, TMR4, and TMR6 are 8-bit timers and are also identical to one another. TMR0 can either be an 8-bit or 16-bit timer.

Configuration of the timers can be done using the MPLAB Code Configurator (MCC) plug-in. The slightly differing Easy Setup windows are shown in Figure 10-2 below. There are four choices to make after you enable the timer. The first choice is which clock frequency to use. The choices are HFINTOSC, FOSC, or FOSC/4. TMR0 does not use FOSC. Recall that FOSC is the same as HFINTOSC when the clock divider is 1. The second choice is the value of the prescaler. Each timer has a different set of possible prescaler values which are possible powers of 2 (see Table 10-1 on next page). The third choice is the value of the postscaler which can be 1 to 16. TMR1 does not have a postscaler. Finally, you request the timer period. Easy Setup for timers shows you the minimum and maximum timer periods you can get in seconds, milliseconds, or microseconds for your choice of prescaler and postscaler. You enter the value followed by s, ms, or us – units matter! Having the minimum and maximum times is useful because we typically use timers for specific time duration. Easy Setup will then determine the *Calculated Period*, the closest timer period to your request value. TMR1 also lets you specify the *period count* as a value between 0 and 65535.

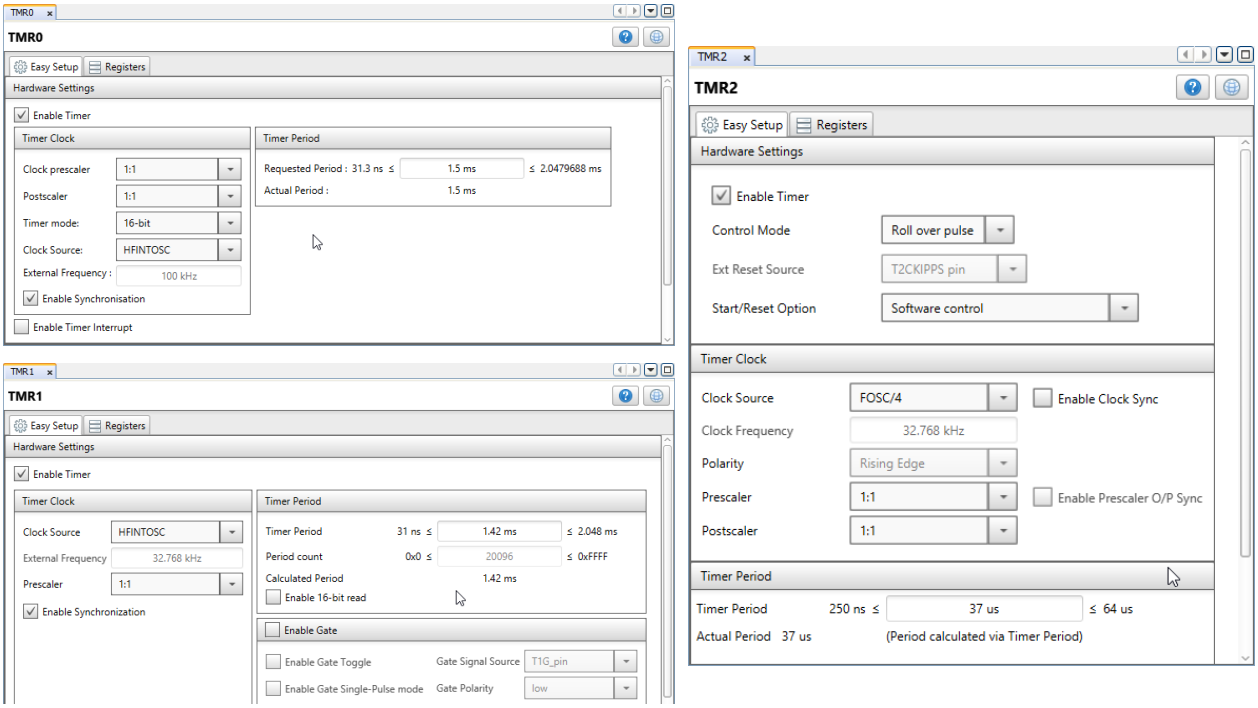



**Figure 10-2:** Timer Easy Setup

**Table 10-1:** Available prescalers and postscalers

| Timer | Prescalers (N) | Postscalers (M) |
|---|---|---|
| TMR0 | 1, 2, 4, 8, …, 2048, 4096, 8192, 16384, 32768 | 1,2, 3, …, 14, 15, 16 |
| TMR1/3/5 | 1, 2, 4, 8 | None |
| TMR2/4/6 | 1, 2, 4, 8, 16, 32, 64, 128 | 1,2, 3, …, 14, 15, 16 |

It is important to understand how the timers reach the time interval you request in Easy Setup. Each 16-bit timer has an internal counter to keep track of clock edges, or multiple of edges if you use a prescaler, until it reaches 65535 when it rolls over to zero and starts counting again much like an odometer on a car. If your postscaler is 1, this sets the timer flag. If your postscaler in greater than one, then the counter must roll over postscaler number of times before the timer flag is set. The maximum you can get is when the counter starts from zero and rolls over to zero. The maximum time is

$$T_{max} = [65536 \times M] \times \frac{1}{f} \times N, \qquad (10\text{-}1)$$

where $1/f$ is the clock frequency being used, $N$ is the prescale value, and $M$ is the postscale value.

The 65536 is in the formula because you are counting from 0 to 65535 which is 65536 counts. The counter does not need to start from zero. To get smaller time intervals, simply start the counter from bigger start values and set the postscaler value to 1. For example, to get the smallest time interval, the start value would be 65535. The timer would only increment once and roll over. The minimum time available would be

$$T_{min} = [1 \times 1] \times \frac{1}{f} \times N. \qquad (10\text{-}2)$$

If you request a time in between, Easy Setup determines a start value for the counter and the time interval would be

$$T = [65536 \times M - startvalue] \times \frac{1}{f} \times N. \qquad (10\text{-}3)$$

This formula arises from the timer incrementing from *startvalue* to 65536, rolling over to zero and incrementing from 0 to 65536 M – 1 more times. Easy Setup will have difficulty giving you small time intervals if the postscaler is greater than one! The minimum period times shown in Requested Period or Time Period in Easy Setup are wrong when the postscaler is greater than one.

For the 16-bit timers, the value is held in two 8-bit SFRs called *TMRxH:TMRxL* where $x$ is 0, 1, 3, or 5. The H in the SFR name is for High byte and the L is for Low byte. You can find the values of these SFRs in the bottom of the *Registers* tab next to the Easy Setup tab as shown in Figure

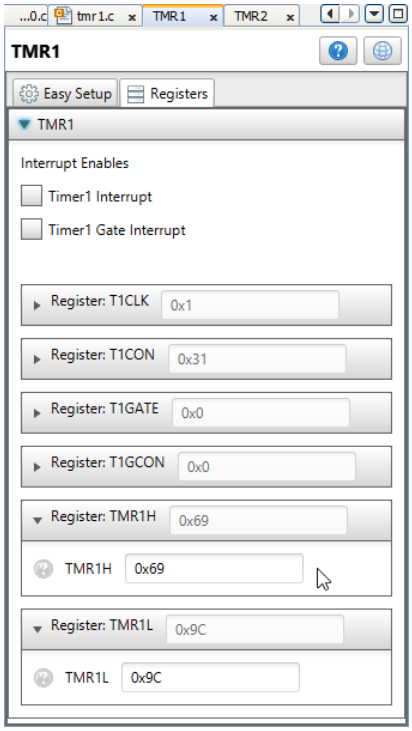



**Figure 10-3:** Timer start value

10-3. Note that hexadecimal or base 16 number 0x69:0x9C, or 0x699C, is decimal 27,036. Try changing *Requested Period* in *Easy Setup* and seeing how these values change under *Registers*.

The 8-bit timers, TMR2/4/6, work a little differently. They can roll over when the counter reaches 256, or $2^8$ as expected, but you can also have them roll over at a smaller number. They do this with an SFR, *TxPR*, where *x* is 2, 4, or 6, that holds that number or period. The counter, another SFR called *TxTMR*, always starts from zero and also resets from zero. When you chose a period in *Easy Setup*, it computes the value of *TxPR* to use and keeps *TxTMR* at zero. The value of these SFRs can be found in the *Registers* tab next to *Easy Setup* as shown in Figure 10-4. Try changing *Time Period* in *Easy Setup* and seeing how these values change under *Registers*. The time you get is

$$T = [TxPR \times M] \times \frac{1}{f} \times N \qquad (10\text{-}4)$$

Generating the MCC code creates libraries of code with function that you can use. For example, there will be a library consisting of tmr0.h and tmr0.c if you use the TMR0 peripheral. Below, in Table 10-2, are the most useful functions from the timer libraries. The use of these functions will be developed in the example that follow.

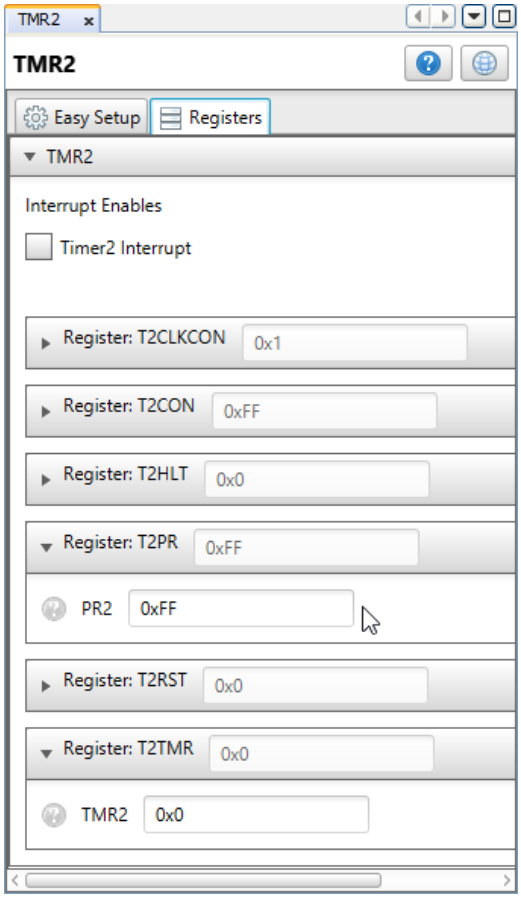



**Figure 10-4:** TxPR and TxTMR values

**Table 10-2: Timer Functions and SFRs**

| Functions, and Variables | Description |
|---|---|
| void TMRx_StartTimer(void) | Starts the timer (counter starts incrementing) if not enabled in Easy Setup |
| void TMRx_StopTimer(void) | Stops the timer, counter stops incrementing. |
| uint16_t TMRx_ReadTimer(void) | Reads the timer start value *TMRxH:TMRxL* or *TxTMR* |
| void TMRx_WriteTimer(uint16_t timerVal) | Changes the value of *TMRxH:TMRxL* with TMR0/1/3/5 or *TxTMR* with TMR2/4/6 |
| void TMRx_Reload(void) | Refreshes the timer with the value of *TMRxH:TMRxL* at initialization. TMR0/1/3/5 only. |
| void TMRx_Period8BitSet(uint8_t periodVal) | Changes the value of *TxPR*. TMR2/4/6 only. |
| TMRx_HasOverflowOccured(void) | Returns value to the flag TMRxIF, which is 1 if the appropriate number of rollovers specified by the postscaler as occurred. |

| | |
|---|---|
| TMRxIF | Flag TMRxIF is 1 if the appropriate number of rollovers specified by the postscaler as occurred. Can be read or written. |
| uint16_t timer0ReloadVal16bit | Value of *TMR0H:TMR0L* at initialization. Can be read or written. |
| uint16_t timerxReloadVal | Value of *TMRxH:TMRxL* at initialization. Can be read or written. TMR1/3/5 only. |
| TxCON1bits.CKPS | Prescaler value. Can be read or written. |
| TxCON0bits.OUTPS | Postscaler value. Can be read or written. |

## Creating Non-blocking Delays

Code such as `DELAY_milliseconds()` and `while(1)` keep the PIC/MCU from responding to input and are called blocking delays. To create a non-blocking delay with the timer peripherals is relatively straightforward. The basic idea is to have a while loop that periodically check the time (more exactly the status of the timer flag) with code inside the while loop to handle other tasks. To use the period chosen in Easy Setup, load the initial value of the timer or use a new value, clear the timer flag and construct your while loop

```
// 16-bit timers TMR0/1/3/5
TMRx_Reload();  // use initial value of TMRxH:TMRxL
TMRxIF = 0;     // clear flag
while(!TMRxIF)  // wait until rollover occurs
{
//code to respond to inputs
}

// 8-bits timers TMR2/4/6
TMRx_WriteTimer(0); // use initial value of TxPR
TMRxIF = 0;         // clear flag
while(!TMRxIF)      // wait until rollover occurs
{
//code to respond to inputs
}
```

The example program `Non-blockingDelays.c` shows several things that you can do during a 30 second delay such as reading text input, or button input, and blinking an LED. Note the line

```
while(!TMR0IF && !IO_RD1_GetValue())
```

which demonstrates how you can combine monitoring of the timer flag with monitoring button status. Also note that the line `while(!TMR0IF);` turns the timer into a blocking delay.

```
// Non-blockingDelays.c
// HFINTOSC = 1 MHz, clock divider = 1, FOSC = 1 MHz
// UART at 9600 bps
// TMR0 PS = 512, Post = 1, period = 30 s
// LED on RD2
// button on RD1
```

```
// delays library

#include "mcc_generated_files/mcc.h"
#include "putty.h"
#include <stdbool.h>
#include <stdio.h>
#include <math.h>

extern volatile unsigned int timer0ReloadVal16bit;  // TMR0H:TMR0L start value

void main(void)
{
    // Initialize the device
    SYSTEM_Initialize();

    unsigned char letter;

    IO_RD2_SetHigh(); // LED on show capture in progress

    clearPuTTY();

    printf("TMR0 nonblocking delays\n\n\r");
    printf("Settings:\n\r");
    printf("Initial value of TMR0H:TMR0L = %u \n\r", timer0ReloadVal16bit);
    printf("Prescaler SFR value = %u , N = %.0f  \n\r", T0CON1bits.CKPS,
            pow(2,T0CON1bits.CKPS));
    printf("Postscaler SFR value = %u , M = %u  \n\n\r", T0CON0bits.OUTPS,
            T0CON0bits.OUTPS + 1);

    printf("30 s Non-blocking delay - please type on your keyboard \n\n\r");
    TMR0_Reload();
    TMR0IF = 0;
    while(!TMR0IF)
    {
      if(UART2_DataReady)
      {
        letter = UART2_Read();    // read a single character
        printf(" You have typed {%c}, ascii #%u \n\r",letter, letter);
      }
    }
    printf("Non-blocking delay finished \n\r");

    printf("30 s Non-blocking delay - blinks LED. Press button to
               end early \n\n\r");
    TMR0_Reload();
    TMR0IF = 0;
    while(!TMR0IF && !IO_RD1_GetValue())
    {
        IO_RD2_Toggle();
        DELAY_milliseconds(20);
    }
    if (TMR0IF == 1)
        printf("Time ran out. Button not pressed.\n\r");
    else
        printf("You pressed button and ended early \n\n\r");

    printf("\nNon-blocking delay finished \n\n\r");
```

```
    printf("\nTimers can make blocking delays \n\n\r");
    TMR0_Reload();
    TMR0IF = 0;
    while(!TMR0IF);
    printf("Blocking delay finished\\n\r");

    IO_RD2_SetLow();         // LED off, timing finished
    while (1);
}
```

## Non-blocking Delays of More than One Duration

Easy Setup is great when you need non-blocking delays of one duration but what if you need several different nonblocking delays? A very simple example would be to blink an LED on for 0.2 seconds and off for 0.35 seconds.

The start of the solution would be to use Easy Setup to find the timer settings for both durations. Beside the timer start value for each, the prescaler and postscaler setting may differ. The code might look like

```
// First duration
TxCON1bits.CKPS = ?;    // set prescaler
TxCON0bits.OUTPS = ?;   // set postscaler
TMRx_WriteTimer(startvalue1);
TMRxIF = 0;     // clear flag
IO_RD2_SetHigh();
while(!TMRxIF)  // wait until rollover occurs
{
//code to respond to inputs
}

// Second duration
TxCON1bits.CKPS = ?;
TxCON0bits.OUTPS = ?;
TMRx_WriteTimer(startvalue2);
TMRxIF = 0;     // clear flag
IO_RD2_SetLow();
while(!TMRxIF)  // wait until rollover occurs
{
//code to respond to inputs
}
```

## Timing Code Execution

Suppose you needed to know how long the line of code `printf("Hello World")` took to execute. Or perhaps you wanted to see how precise `DELAY_millisceconds(100)` is. Both and much more can be determined by timers. You need to have a ballpark estimate for how long the code takes to execute. Setting up a possible 10 second timer for something that is only 15

milliseconds long lowers the precision of your result. Not to worry, you can always redo the timing measurement with a smaller timer duration setting.

The format for timing code would be

```
n = TxCON1bits.CKPS;
m = TxCON0bits.OUTPS; // postscaler = 1 for timing code
TMRx_WriteTimer(0);
TMRxIF = 0;     // clear flag
//
// code to time in here
//
flag = TMRxIF; // if = 1, timer rolled over. Beware!
count = TMRx_ReadTimer();
if (flag == 0)
   {// no rollover
    Time = count * 1/freq * N;
   }
else
   {
    // Undetermined time, increase prescaler or decrease freq
   }
```

Always use a postscaler equal to 1 for times. If this case, when the flag is zero, you are guaranteed that the timer has not rolled over. Below is example code using TMR0.

```
// timeCodeExecution.c
// 16bitTimersAsDelays.c
// Generate MCC so that the FOSC = 32 MHZ and Delays
// TMR0 output on RC0 and LED on RD2 (led_RD2)

#include "mcc_generated_files/mcc.h"
#include <stdio.h>    // C Library for printf()
#include <math.h>     // C Library for pow(x,n)
#include <stdbool.h>
#include "putty.h"

extern volatile uint16_t timer0ReloadVal16bit;

int main(void)
{
 // Initialize the device
 SYSTEM_Initialize();

 bool flag = 0u;
 unsigned int counter1, counter2, counter3;
 unsigned int n, m=0;
 float time;

 clearPuTTY();                    // clear display
 printf("Timing Code Execution \n\n\r");

 n =  T0CON1bits.CKPS;
 m =  T0CON0bits.OUTPS;
 printf("Initial settings: TMR0H:TMR0L = %u prescaler = %u postscaler
```

```
           = %u\n\n\r", timer0ReloadVal16bit, n, m);

 time = (65536.0)*pow(2.0,n)*(m+1.0)*4.0/_XTAL_FREQ;
 printf("Initial settings: FOSC = %.0f, tmax = %f seconds\n\n\r",
        (float)_XTAL_FREQ, time );

 // This is simple code to check if counter is incrementing
 TMR0IF = 0;
 TMR0_WriteTimer(0);
 counter1 = TMR0_ReadTimer();
 counter2 = TMR0_ReadTimer();
 counter3 = TMR0_ReadTimer();
 printf("Readings 1 = %u, 2 = %u, 3 = %u \n\n\r", counter1, counter2,
                           counter3);

 //Timing various delays for comparison
 TMR0IF = 0;                     // clear flag
 TMR0_WriteTimer(0);             // reset counter
 DELAY_milliseconds(30);         // can time single or multiple lines of code
 flag = TMR0IF;
 counter1 = TMR0_ReadTimer();
 time = (float)counter1*pow(2.0,n) * 4.0/ _XTAL_FREQ;
 printf("wait =  30 ms: counter = %5u, flag = %u (if = 1, time overflowed),
         t = %f sec \n\n\r", counter1, flag, time);

 TMR0IF = 0;                     // clear flag
 TMR0_WriteTimer(0);             // reset counter
 DELAY_milliseconds(70);         // can time single or multiple lines of code
 flag = TMR0IF;
 counter1 = TMR0_ReadTimer();
 time = (float)counter1*pow(2.0,n) * 4.0/ _XTAL_FREQ;
 printf("wait =  70 ms: counter = %5u, flag = %u (if = 1, time overflowed),
          t = %f sec \n\n\r", counter1, flag, time);

 TMR0IF = 0;                     // clear flag
 TMR0_WriteTimer(0);             // reset counter
 DELAY_milliseconds(150);        // can time single or multiple lines of code
 flag = TMR0IF;
 counter1 = TMR0_ReadTimer();
 time = (float)counter1*pow(2.0,n)*(m+1.0) * 4.0/ _XTAL_FREQ;
 printf("wait = 150 ms: counter = %5u, flag = %u (if = 1, time overflowed),
        t = %f sec \n\n\r", counter1, flag, time);

 TMR0IF = 0;                    // clear flag
 TMR0_WriteTimer(0);            // reset counter
 DELAY_milliseconds(300);      // can time single or multiple lines of code
 flag = TMR0IF;
 counter1 = TMR0_ReadTimer();
 time = (float)counter1*pow(2.0,n)*(m+1.0) * 4.0/ _XTAL_FREQ;
 printf("wait = 300 ms: counter = %5u, flag = %u (if = 1, time overflowed),
         t = %f sec \n\r", counter1, flag, time);

 while(1);
}
```

## Timing Events

Suppose you want to know how long it takes you to type 10 letters on the keyboard. You would need code something like this

First letter detected;

```
TMRx_WriteTimer(0);
```

Tenth letter detected

```
flag = TMRxIF; // if = 1, timer rolled over. Beware!
count = TMRx_ReadTimer();
```

Always use a postscaler equal to 1 for times. If this case, when the flag is zero, you are guaranteed that the timer has not rolled over. The following example check to see if you can press a button ten times in under 2 seconds.

```
// timeEvent.c
// FOSC = 32 MHz
// TMR0 uses FOSC/4 max period 2 seconds, M = 1
// button_RD1

#include "mcc_generated_files/mcc.h"
#include <stdio.h>    // C Library for printf()
#include <math.h>     // C Library for pow(x,n)
#include <stdbool.h>
#include "putty.h"
#include "buttons.h"

extern volatile uint16_t timer0ReloadVal16bit;

int main(void)
{
 // Initialize the device
 SYSTEM_Initialize();

 unsigned int counter = 0, value = 0, flag = 0, m, n;
 float time;
 unsigned char has_switch1_changed = 0;

 clearPuTTY();
 printf("Time Ten Button Presses Under 2 Seconds \n\n\r");

 n = T0CON1bits.CKPS;   // prescaler setting n - prescaler is N = 2^n
 m = T0CON0bits.OUTPS;  // postscaler setting m - postscaler is M = m + 1
 time = 65536.0*4.0*pow(2,n)/(float)_XTAL_FREQ;
 printf("timer0ReloadVal16bit %u = 0x%x, n = %u, N = %.0f, m = %u, M = %u
         tmax = %f\n\n\r", timer0ReloadVal16bit, timer0ReloadVal16bit, n,
         pow(2,n), m, m+1, time);

 while(1)
 {
   has_switch1_changed = poll_switch1_for_edges(button_RD1_GetValue());

   if ( has_switch1_changed == 1 ) //rising edge
     {
```

```
      counter++; // increment button counter
      DELAY_milliseconds(5); // debouncing delay
      if (counter == 1) // start the timer on first press
        {
         TMR0_WriteTimer(0);     // reset counter
         TMR0IF = 0;             // clear flag
        }
      if (counter == 10)         // read the timer on 10th press
        {
         value = TMR0_ReadTimer();
         time = (float)value*4.0*pow(2,n)/(float)_XTAL_FREQ;
         flag = TMR0IF;
         if (flag == 1)
           {
            printf("t > 2 seconds, count rolled over \n\r");
           }
         else
           {
            printf("t = %f (count = %u, flag = %u)\n\r",time, value, flag);
           }
          counter = 0; //reset
         }
     } //end if

 } //end while
}// timeEvent.c
```

## Timers as Counters

Timers measure time by counting edges, rising or falling, of square waves with a well-known period. The time measurement is simply the number of edges multiplied by the period. The square waves can come from an internal or an external clock source. Counting, by itself, is a basic useful measurement. If what you wish to measure can be arranged to generate pulses separated in time, the timer peripheral can count those pulses. Imagine, for instance, you wanted to know the number of people entering a doorway. A turnstile and a circuit to generate a square pulse at each turn of the turnstile is common solution.

Configuring the timer for external counting is straightforward in Easy Setup. In Figure 10-2, the *Clock Source* dropdown menu has many items besides HFINTOSC, FOSC, or FOSC4. To enable counting on an external pin, the choices are *T0CKI_PIN* for TMR0 and *TxCKIPPS* for all the other timers. Also, the prescalers and postscalers should be set to 1.You also need to visit the Pin Manager: Grid View and assign physical pins (Figure 10-5). For TMR0/1/3/5, the external pins are named *TxCKI* and for TMR2/4/6 they are named *TxIN*.

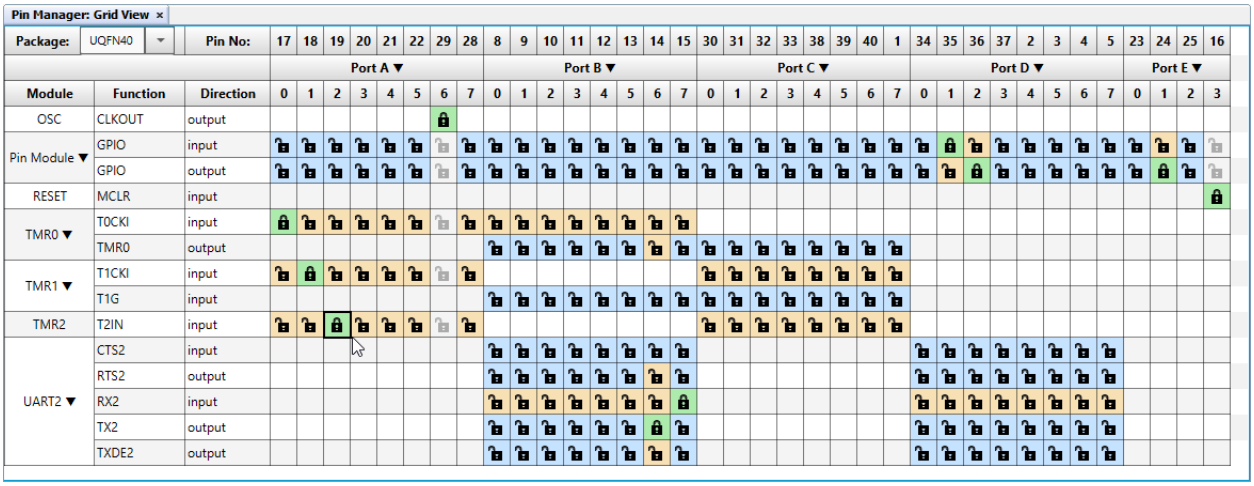



**Figure 10-5:** Timer external pin selection

The following example show how to use a timer counter to count button presses. The button mus be wited to the timer input pin.

```
// tmr0Counter.c
// FOSC = 32 MHz
// TMR0 uses Ext edge. N = M = 1
// Set pin T0CKI = RA&
// Attach button to RA7/T0CKI

#include "mcc_generated_files/mcc.h"
#include <stdio.h>     // C Library for printf()
#include <math.h>      // C Library for pow(x,n)
#include <stdbool.h>
#include "putty.h"

extern volatile uint16_t timer0ReloadVal16bit;  // defined in tmr0.c

int main(void)
{
 // Initialize the device
 SYSTEM_Initialize();

 int i;
 unsigned int newcount = 0, oldcount = 0;

 clearPuTTY();                  // clear display
 printf("Test Timer as Counter \n\r");

 printf("Initial settings: TMR0H:TMR0L = %u prescaler = %u postscaler =
        %u\n\n\n\r",timer0ReloadVal16bit, T0CON1bits.CKPS, T0CON0bits.OUTPS);

 // counter range is 0 to 10
 TMR0IF = 0;            // clear flag
 TMR0_WriteTimer(0);              // reset counter

 printf("Ready \n\n\r");
 while(1){   // only shows 10 presses
```

```
      newcount = TMR0_ReadTimer();
      DELAY_milliseconds(10);
      if (newcount > oldcount){ // print only on change
          if( newcount > (oldcount + 1) )
          {
              newcount = oldcount + 1;
              TMR0_WriteTimer(newcount);
          }

          printf("Count = %u \n\r", newcount);
          oldcount++;
         }
      if (newcount >= 10)
       {
          TMR0_WriteTimer(0); // reset count to 0
          oldcount = 0;
       }
      }
} // end program
```

# Chapter 11. Pulse-Width Modulation (PWM)

Generating a continuous square wave signal and varying either the period or duty cycle, see Figure 11-1, is a common way to control DC devices such as motors, LEDs, and speakers. The on time, $T_{on}$, is often referred to as the *pulse width* or the *duty cycle*. The ratio of $T_{on}$ to the period $T$ is often expressed as a percentage and called the *% Duty Cycle*. This sort of control signal is called Pulse Width Modulation (PWM) because you modulate, or control, the pulse width. The PIC18F46K42 has eight separate modules for PWM, four CCPx peripherals with x being 1, 2, 3, or 4 and four PWMx peripherals where x = 5, 6, 7, or 8. The CCPx peripherals are multifunction and PWM is only one of several different capabilities. You can "set and forget" a square wave signal on one of these peripherals and assign an accompanying output pin. The PWM signal is then generated continuously from that pin with no need of further intervention on your part. This allows you to do other things at the same time.

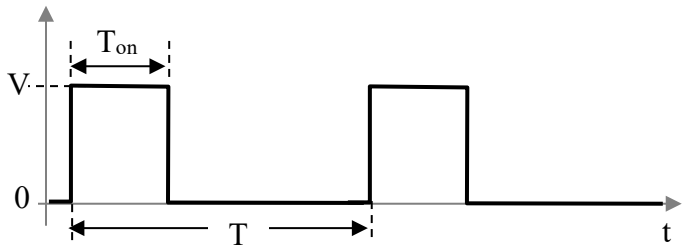



**Figure 11-1:** Square wave signal.

The PWM peripherals work on top of the timer TMR2/4/6 peripherals. The timers set the frequency or period of the signal. Note that the postscaler settings do not work in PWM. If you set a postscaler to something other than one, you will generate a warning notification that the postscaler setting does nothing. As a result, you must configure your timer in the MPLAB Code Configurator (MCC) plug-in before the PWM peripheral. You must choose *Clock Source* to be *FOSC/4*; any other setting generates a warning notification. You can have any allowable value of *Prescaler* and *Timer Period* as in Figure 11-2. The PWM period is set by the formula

$$T_{PWM} = [PRx + 1] \times N_{prescaler} \times \frac{4}{FOSC} \quad \textbf{[ 11-1 ]}$$

where *PRx* is the period Special Function Register (SFR) for TMRx and which is an integer between 0 and 255.

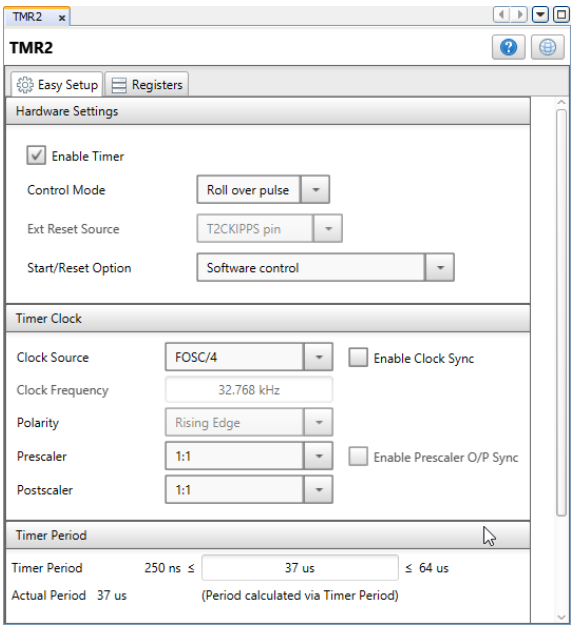



**Figure 11-2:** Timer Settings for PWM.

The choices in the configuration of the CCPx and PWMx are identical except that in the MCC Easy Setup for CCPx, you must first choose the PWM function of the peripheral. The only settings to make in CCPx or PWMx are to select which of TMR2/4/6 to use and to set the % *Duty Cycle* as shown in Figure 11-3 and Figure 11-4. Setting the %DC generates the value of the *Duty Cycle Value* SFR which is called *CCPRx* and *PWMxDC*, respectively, which are shown below the % Duty Cycle. The duty cycle value is a integer between 0 and 1023.

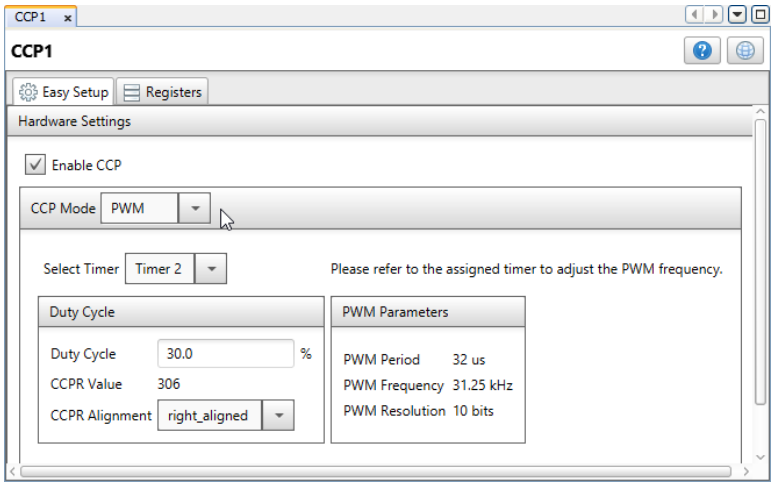



**Figure 11-3:** % Duty Cycle and value.

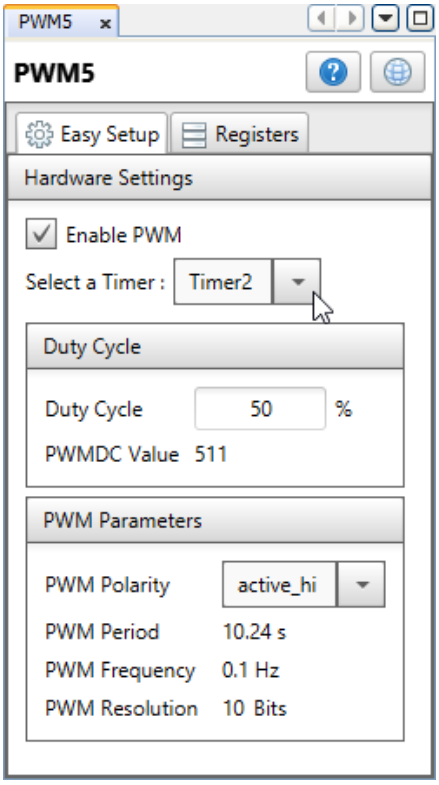



**Figure 11-4:** %DC and value.

The on time is set by the formula

$$T_{dutycycle} = dutycycleValue \times N_{prescaler} \times \frac{1}{FOSC} \qquad \textbf{[ 11-2 ]}$$

And the % Duty Cycle is

$$\%DC = \frac{dutycycleValue}{[PRx+1]\times 4} \qquad \textbf{[ 11-3 ]}$$

Since *PRx* can be a value less than 255, there are values of *CCPRx* and *PWMxDC* which would make the %DC greater than 100%. However, you cannot have a square wave that is on more than 100% of the time.

The last step is to assign a pin in Pin Manager: Grid View to output the CCPx or PWMx signals as shown in Figure 11-5. Also check the Pin Module, Figure 11-6, and make that the *Analog* boxes are unchecked since PWM is a digital signal.

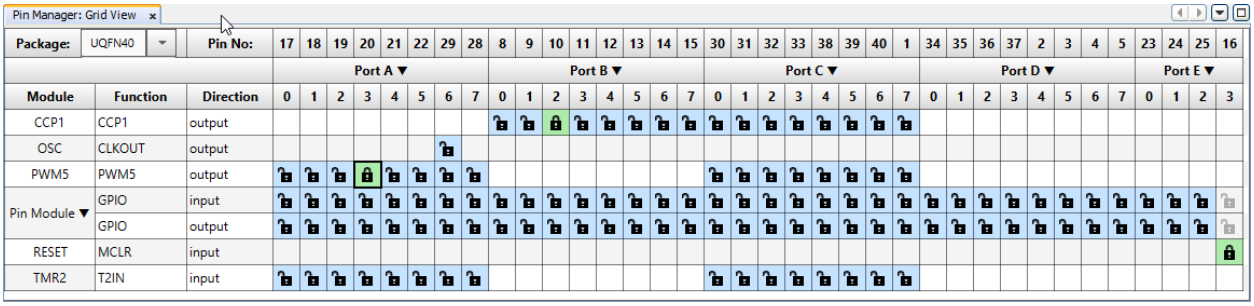



**Figure 11-5:** Pin selection for PWM signal.

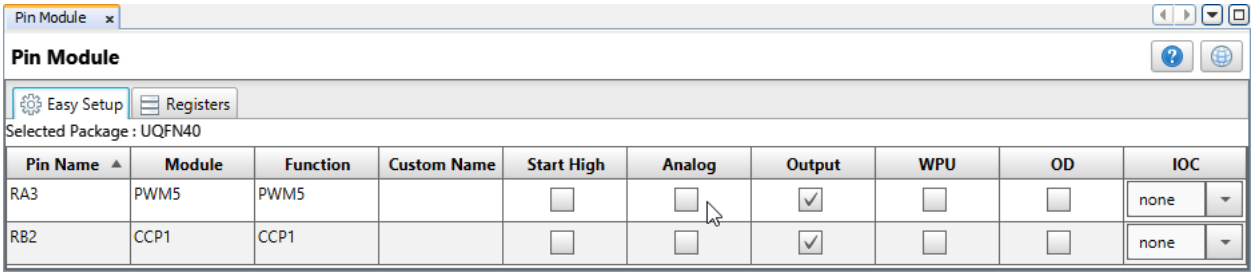



**Figure 11-6:** PWM pin should be digital.

Now when the function `SYSTEM_Initialize()` is called, the output pins automatically produce the square waves.

## PWM and LEDs, Motors, and Speakers

An LED will light up when the voltage is on and be dark when the voltage is low. If a square wave with a long period drives the LED, you see the LED blink on and off. At smaller periods, for frequencies more than about 20 Hz, the blinking is too fast for your eyes to separate and the light level appears constant though dimmer than full bright since it is part on and part off. Thus, a small duty cycle means a dim light and a large duty cycle means a bright light. PWM is thus the equivalent of a brightness control for an LED.

For a motor, PWM is a way of controlling motor speed. An ideal motor would have no inertia and instantly turn on or off as shown in Figure 11-7(a). Real motors have a certain amount of inertia. When the motor is turned on, and the period is long enough, it takes some time to reach full speed as shown in as shown in Figure 11-7(b). When turned off, the motor also takes some time to slow to a stop. For a short-period signal, however, as shown in Figure 11-7(c), a motor will never reach full speed during the on cycle and never completely slow during the off cycle. Instead it turns at some speed between zero and maximum. The ripple in the speed as shown in Figure 11-7(c) is exaggerated for effect and is not noticeable at small enough periods. By controlling the duty cycle, the motor can have many speeds between full off and its tops speed.

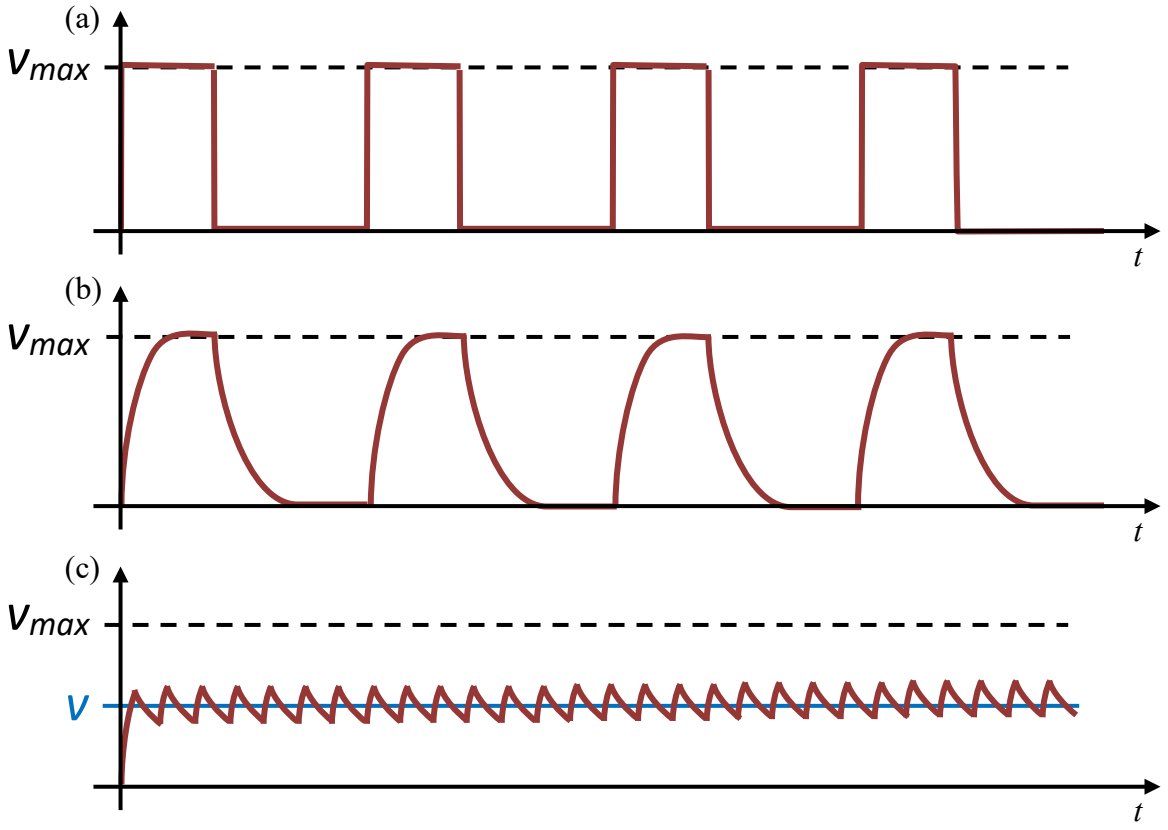



**Figure 11-7:** Effect of inertia and period on motors.

If you are interested in controlling either LED brightness or motor speed, you should choose an appropriately small period and keep it fixed. Brightness or speed are then changed changing the duty cycle. PWM is an efficient method to control brightness and speed because energy is not wasted during the off cycle and is widely used as a result. For comparison's sake, consider an alternate control circuit consisting of a variable resistor in series with the LED or motor. When the resistance is low, the current is high, and the LED is bright and the motor spins fast. When the resistance is high, the current is low, and the LED is dim and the motor spins slowly. This is not a very efficient approach because energy is wasted in heating the resistor.

DC motors are more complicated than LEDs, in that they can operate both forward and in reverse, normally draw a lot of current, and can drive a large current backward into the signal source when it switches on and off (*back emf)*. Consequently, to protect the MCU, a separate chip called a *driver* is usually put between the MCU and a motor. The driver can supply much bigger currents than the MCU and is designed to withstand much greater reverse currents. At the very least a diode should be used in series with a motor to keep currents from entering the output pin when it is in its low state.

Speaker control typically refers to loudness or frequency. PWM is only effective for frequency. When we change the period, the frequency is different, and we hear a different tone. The duty

cycle is kept constant at 50% since the duty cycle hardly affects what we hear except when it is near zero or 100%.

## PWM Functions

You have access to the initial value duty cycle SFR through the defined constant `PWMx_INITIALIZE_DUTY_VALUE` for the PWM modules. For the CCPx modules, you must use the SFR `CCPR1H:CCPR1L`. You can change that value of the duty cycle at any time using the MCC supplied function `PWMx_LoadDutyValue(dutyvalue)`. Remember to use only values between 0 and 1023. A value of zero turns the square wave off. The PWM period can also be changed by the MCC timer functions described in *Chapter 10 - Timers.*

PWM modules using different timer modules can have different frequencies but PWM modules using the same module have the period. When two PWM peripherals use the same timer, and the second PWM peripherals is enabled sometime later than the first, the rising edge of the two pulse widths start at the same instant as shown in Figure 11-8

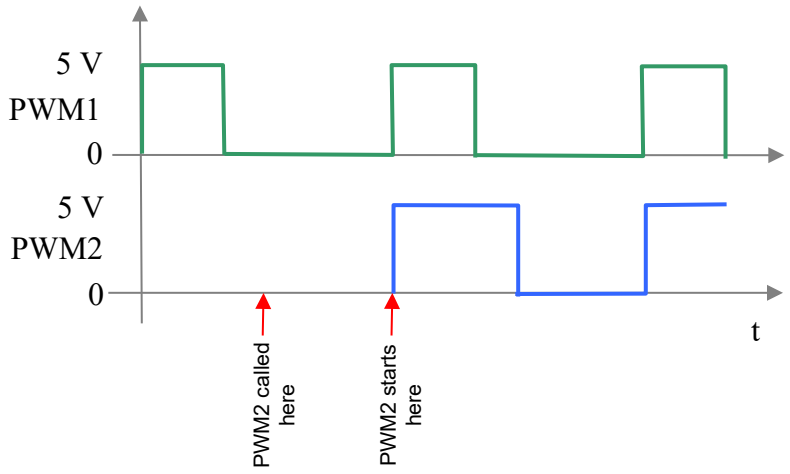



**Figure 11-8:** The rising edge of the two PWM signals always coincide.

| Q. At 32 MHz, what is the longest period a PWM output can have? What is the shortest? For the shortest period, what duty cycles are possible? |
|---|

A short example of how to use the MCC PWM and TMR functions in a program is given below. Connect a button to RD1 and an LED to RD2. Pressing the button changes the brightness of the LED.

```
// PWM.c
// FOSC = 1 MHz
// CCP1 using TMR2, DC = 50%, RC5 and RC4
// TMR2 uses FOSC/4. N = 128, M = 1, use maximum period
```

```
// PWM5 using TMR4, DC = 30%, RC7
// TMR4 uses FOSC/4. N = 64, M = 1, use maximum period
// button on RD1, LED on RD2
#include "mcc_generated_files/mcc.h"
#include <stdio.h>    // C Library for printf()
#include <math.h>     // C Library for pow(x,n)
#include "buttons.h"
#include "putty.h"

void main(void)
{
    SYSTEM_Initialize();
    clearPuTTY();

    unsigned int n = 0, m = 0, counter = 0, has_switch1_changed = 0;
    float T, f;

    n = T2CONbits.CKPS;     // prescaler setting
    T = ((float)PR2 + 1.0)*pow(2.0,n)*4.0/_XTAL_FREQ;
    f = 1/T;
    printf("TMR2 settings: %u = 0x%x, N = %.0f, T = %f s, f = %f Hz \n\r",
           PR2, PR2,pow(2,n), T, f);

    //Duty cycle value stored in SFRs CCPR1H:CCPRxL
    printf("Easy Setup value of CCPR1 is %4.0f \n\r", CCPR1H*256.0+CCPR1L);
    printf("%DC = %.1f \n\n\r",(CCPR1H*256.0+CCPR1L)*100.0/(PR2+1)/4.0);


    n = T4CONbits.CKPS;     // prescaler setting
    T = ((float)PR4 + 1.0)*pow(2.0,n)*4.0/_XTAL_FREQ;
    f = 1/T;
    printf("TMR4 settings: %u = 0x%x, N = %.0f, T = %f s, f = %f Hz \n\r",
           PR4, PR4,pow(2,n), T, f);

    //Duty cycle value stored in SFRs PWMxDCH:PWMxDCL
          (or PWM5_INITIALIZE_DUTY_VALUE)
    printf("Easy Setup value of PWM5DC is %u \n\r",
             PWM5_INITIALIZE_DUTY_VALUE);
    printf("%DC = %.1f \n\n\r",
            (float)PWM5_INITIALIZE_DUTY_VALUE*100.0/(PR4+1)/4.0);


    printf("counter = %u OFF\n\r",counter);

    while (1)
    {
      has_switch1_changed = poll_switch1_for_edges(IO_RD1_GetValue());
      if ( has_switch1_changed == 1 )
      {
          counter++;
          if (counter > 5) counter = 0;
          printf("counter = %u \n\r",counter);
      }

    switch(counter){
        case 0: PWM1_LoadDutyValue(0);
                PWM5_LoadDutyValue(0);
```

```
                    break;
            case 1: PWM1_LoadDutyValue(128);
                    PWM5_LoadDutyValue(128);
                    break;
            case 2: PWM1_LoadDutyValue(256);
                    PWM5_LoadDutyValue(256);
                    break;
            case 3: PWM1_LoadDutyValue(511);
                    PWM5_LoadDutyValue(511);
                    break;
            case 4: PWM1_LoadDutyValue(768);
                    PWM5_LoadDutyValue(768);
                    break;
            case 5: PWM1_LoadDutyValue(1023);
                    PWM5_LoadDutyValue(1023);
            default:
                    break;
        }

        }
}
```

# Chapter 12. Capture

The four capture modules of the PIC MCU provides the ability to time when falling (high to low) or rising (low to high) edge voltage signals occur at designated capture digital input pins named ICPx where x = 1, 2, 3, or 4. Capture, along with PWM discussed in the previous chapter, are functions of the CCPx peripherals. The Capture modules depend on the 16-bit timer modules TMR1/3/5.

The typical input signals you might want to measure include the duration of a single pulse which might come from a single button press or a flag passing through a photogate, the pulse width and period of a continuous train of identical square waves which might be timing information from a device, or the on and off times for a train of arbitrary pulses such as when a picket fence falls through a photogate. Plots of such signals are shown in Figure 12-1 below.

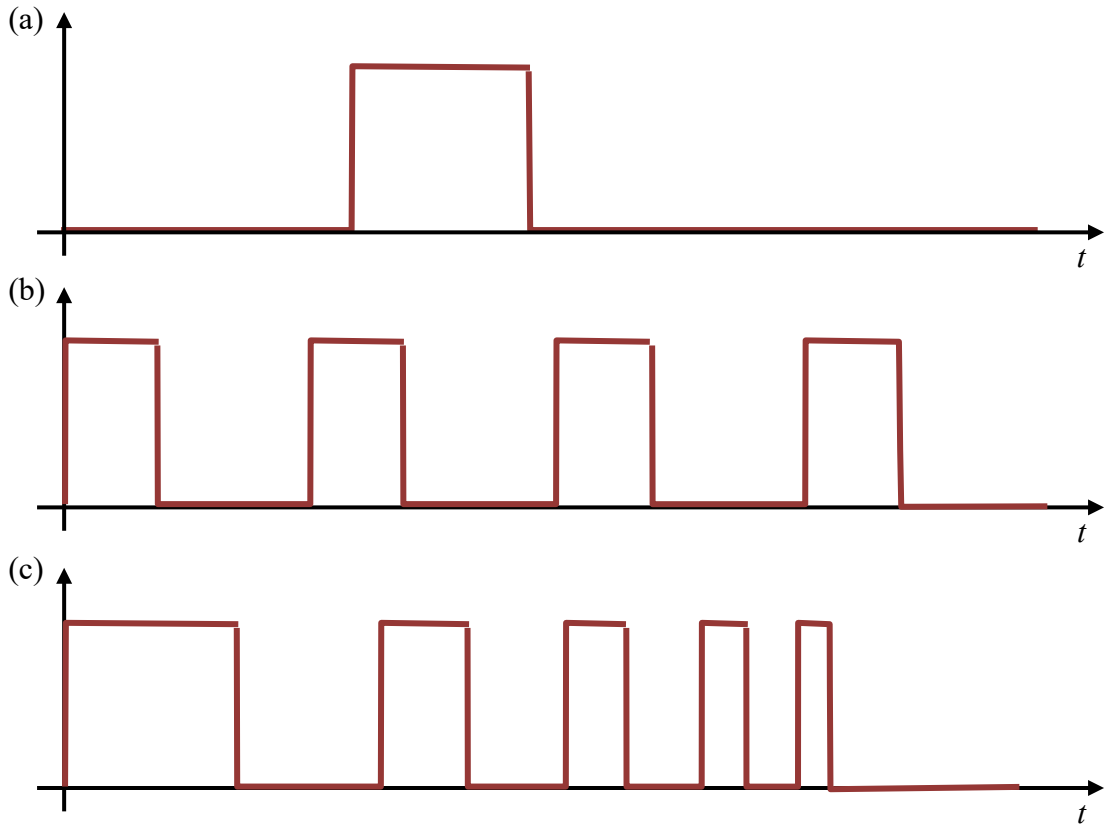



**Figure 12-1:** Typical digital input: (a) pulse, (c) continuous square wave, and (c) arbitrary.

For the single pulse, one need only capture a rising edge and a subsequent falling edge to time the pulse width. For the continuous square wave, one needs at least three consecutive edges to determine pulse width and period. A cautious person would capture more and get several pulse width and period measures to check for consistency. For the arbitrary signal, one would most likely know the number of edges that will occur and would want to capture them all.

Capture works by looking for the edges in an input signal. When the edge occurs, the module adds the time of the next rising edge of the associated timer module to a buffer and sets the capture flag to let you know data is available. The module can distinguish between rising edges and falling edges. So, it can measure only falling edges, rising edges, or both. It is important to read the buffer before the next edge occurs because, the new value will overwrite the old value. The behaviour is shown in Figure 12-2 where the signal to be measured is on the bottom and the internal timer signal is on top. The module is only looking for rising edges, and ignores the fallings edges labelled as *A*. When the first rising edge occurs, labelled *1* in the diagram, the capture buffer is filled with the time at the next rising edge of the timer. This would be $t_1 = 9$. When the next rising edge occurs, labelled *2* in the diagram, the capture buffer is filled with the time at the next rising edge of the timer. This would be $t_2 = 8$. Because the timer in the diagram has a period of 10, the timer has rolled over in the time between measurements and the `TMRx_IF` has been set at the time labelled *D*. In most cases, you would be using a large period (typically 65536) and suitable prescaler value, so that rollover does not occur. Remember with rollover, you may have no way of knowing whether a timer has rolled over more than once between measurements.

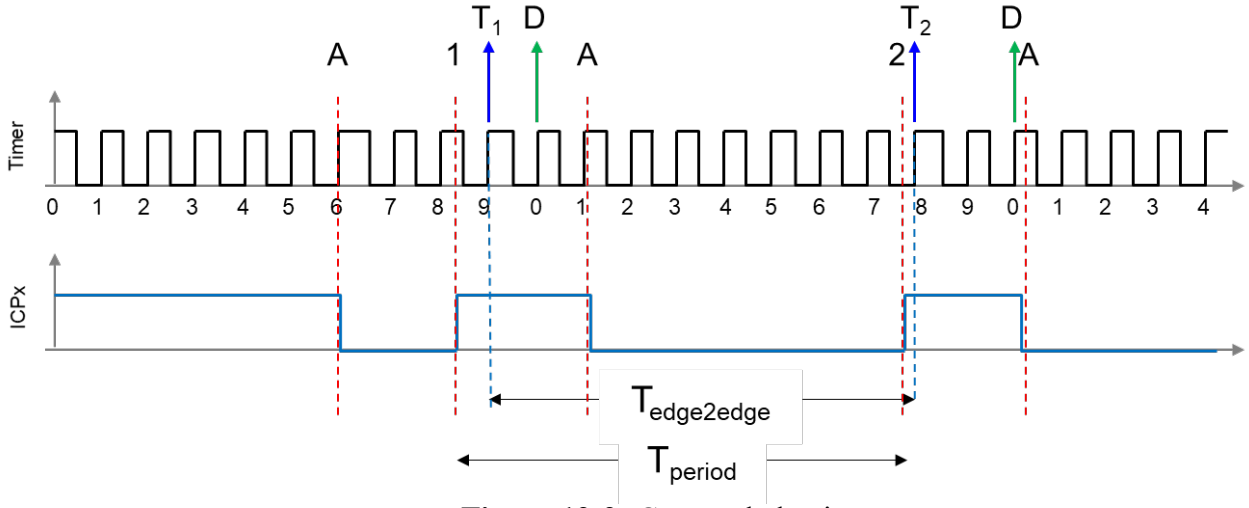



**Figure 12-2:** Capture behaviour.

Note that the capture data lets you calculate the time labelled $T_{edge2edge}$ in the diagram but that this is slightly different from the actual $T_{period}$ of the signal. In Figure 12-2, the difference is about one period of the timer. If you did get such a small time for the period, you should switch your system to a higher frequency. When the timer does not overflow, the time between edges will be determined by the formula

$$T_{edge2edge} = [T_2 - T_1] \times N \times \frac{1}{f} \qquad [12\text{-}1]$$

where $N$ is the timer prescaler and $f$ is the frequency of the timer.

Configuration in the MPLAB Code Configurator (MCC) plug-in is straightforward. Since capture depends on a timer, select and configure one of TMR1/3/5 first. In Easy Setup, be sure to check the boxes *Enable Timer* and *Enable 16-bit Read* as shown in Figure 12-3. *Clock Source* must be set to FOSC/4. Other settings generate a warning message in the Notifications window. Choose

any *Prescaler* and *Time Period* or *Period Count* but ensure capture events occur before the timer rolls over. It is wise to zero the timer flag `TMRxIF` before capture begins and to check it afterwards for rollover.

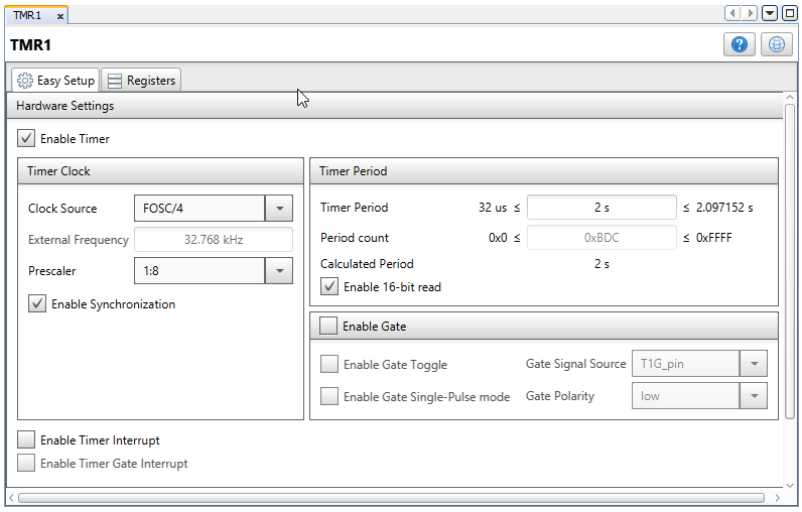



**Figure 12-3:** Timer settings for Capture.

To configure the CCPx peripheral in MCC, first select the *Capture* function in Easy Step as shown in Figure 12-4. Next indicate which timer to use in *Select Timer*. Your choice of *Capture Mode* depends on what you are trying to measure. The choices as shown are every $4^{th}$ *rising edge*, every $16^{th}$ *rising edge*, every *Rising edge*, every *Falling edge*, and *Every edge* both falling and rising. You might use the $4^{th}$ *rising edge* and $16^{th}$ *rising edge* to determine the period of a continuous square wave of high frequency only somewhat smaller than your timer frequency. By measuring every $N^{th}$ edge and dividing by $N$, you will get a more precise and accurate answer.

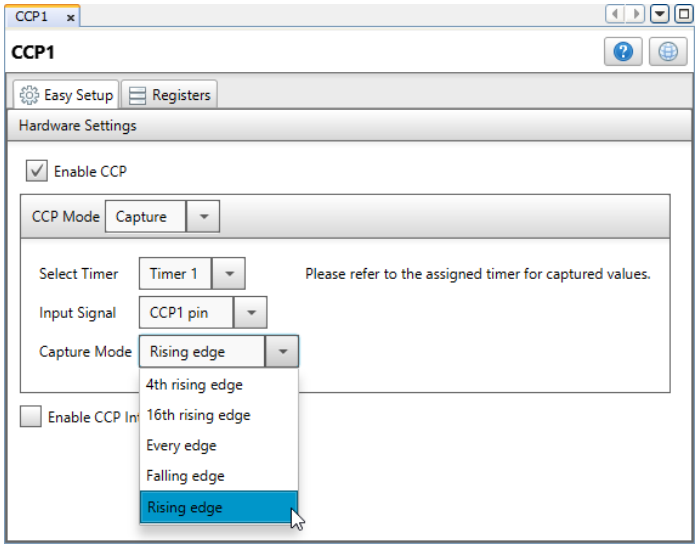



**Figure 12-4:** Capture settings.

Using the *Rising edge* or the *Falling edge* Capture modes to determine the period of continuous square wave would be fine as long as the frequency of the signal is significantly smaller than the frequency of the timer. If you need to calculate the pulse width or on time of a pulse, you will need the *Every edge* Capture mode. The *Every edge* mode has a possible ambiguity if you use it on a square wave signal and want to measure the pulse width and the period. You would need to capture at least three consecutive edges to determine pulse width and period. Unless you somehow know that the signal starts with a rising or falling edge, you could have the pair of ambiguous case shown in Figure 12-5, you can accurately determine the period but will be unsure. Later we will see how to handle this situation.

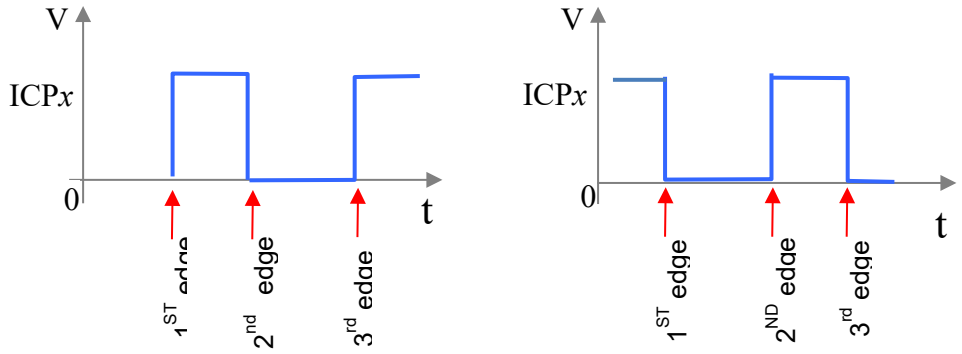



**Figure 12-5:** Timing ambiguity

The last choice is to use Pin Manager: Grid View to designate an input pin aa ICPx as shown in below Figure 12-6 where RC2 is chosen.

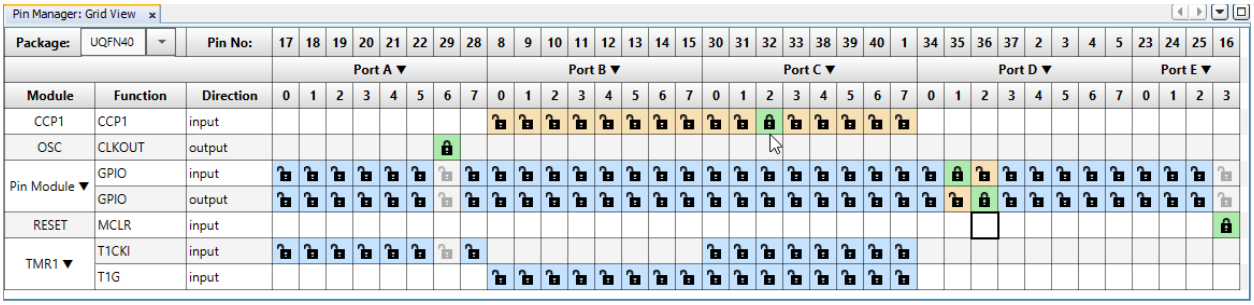



**Figure 12-6:** Capture input pin selection.

When MCC is used to configure the CCPx pins for capture, it generates a function useful for reading the times in the capture buffer. This function, `CCPx_CaptureRead()`, returns an unsigned int with the value of the buffer. To know when to read the buffer, there is a macro `CCPx_IsCapturedDataReady()` that returns the value of the capture flag `CCPxIF`. You can use a Special Function Register, `CCPxCONbits.MODE`, to read or change the capture mode during the execution of your programs. See Table 12-1 for the capture mode settings. The timer period can also be changed by the MCC timer functions described in *Chapter 10 - Timers.*

**Table 12-1:** Capture modes

| Functions and SFRs | Description |
|---|---|
| CCPx_IsCapturedDataReady() | Has an edge been timed? |
| CCPx_CaptureRead() | Read the timer value of captured edge. |
| CCPxIF | Read/Write. Capture flag = 1, when an edge is found. O otherwise |
| CCPxCONbits.MODE | Read/Write. Capture mode.<br>Value Mode<br>7 Every 16th rising edge<br>6 Every 4th rising edge<br>5 Every rising edge<br>4 Every falling edge |

The example code below shows how to use `CCPxCONbits.MODE` to avoid the ambiguity discussed above. If you first use the rising edge mode, capture that first rising edge, and then switch modes to looking for either a falling edges or every edge, you resolve the ambiguity by knowing that the first edge is rising. Switching between capture modes does take a little time, so this is not appropriate for very short pulse widths.

The example code below captures three edges. To provide a signal to capture, the code generates a PWM signal output on pin RC5. The capture input pin is RB0. The pins RB0 and RC5 must be wired together. MCC is configured for finding rising edges. The timer is zeroed, and the timer and capture flags are cleared before searching for edges. This minimizes the change that the timer rolls over between captures. The program then waits for a rising edge to come along using `while(!CCP1_IsCapturedDataReady());`. When found, the value of the timer held in the capture buffer is immediately read into permanent memory. The capture mode is then set to find falling edges and the code waits for that edge. When a falling edge is caught and saved, the capture mode is set back to rising edges. After the third edge is found, The time between edges is found using the formula [12-1].

```
// 16bitCapture.c
// FOSC at 4 MHz, External Oscillator for accuracy
// UART at 9600 bps
// TMR2 using FOSC/4 at Prescaler = 128, period = 32 ms
// PWM5 at 249 / 4*128*4/FOSC = 7.986 ms (DC = 25%), output pin is RC5
// PWM5 is signal we wish to capture
// Start button on RD1
// LED on RD2
// TMR1 using FOSC/4, Prescaler = 1, 16bit period = 65 ms
// CCP1 in Capture Mode, every rising edge, using TMR1, input pin is RBO
//
// Want to capture T_ON and T_PWM
// Note:
// CCP1CONbits.Mode 7 Every 16th rising edge
//                  6 Every 4th rising edge
//                  5 Every rising edge
//                  4 Every falling edge
//                  3 Every edge

#include "mcc_generated_files/mcc.h"
#include "putty.h"
#include <stdbool.h>

void main(void)
{
```

```
    // Initialize the device
    SYSTEM_Initialize();
    uint16_t capture1, capture2, capture3;
    unsigned int flag;
    float T_ON, T_PWM;

    while(IO_RD1_GetValue() == 0); //wait for button press;
    IO_RD2_SetHigh(); // LED on show capture in progress
    clearPuTTY();

    printf("Capture test \n\n\r");

    TMR1_WriteTimer(0);   // we want small numbers if possible
    TMR1IF = 0;           // reset overflow flag
    CCP1IF = 0;           // reset capture flag
    while(!CCP1_IsCapturedDataReady());   //Used only for polling method
    capture1 = CCP1_CaptureRead();  // First rising edge

    CCP1CONbits.MODE = 4;  // switch mode to catch Falling Edge next
    while(!CCP1_IsCapturedDataReady());   //Used only for polling method
    capture2 = CCP1_CaptureRead();  // First falling edge

    CCP1CONbits.MODE = 5;  // switch mode to catch Rising Edge next
    while(!CCP1_IsCapturedDataReady());   //Used only for polling method
    capture3 = CCP1_CaptureRead();  // Second rising edge

    // check if TMR1 has overflowed
    flag = TMR1_HasOverflowOccured();

    printf("T1 = %u , T2 = %u, T3 = %u \n\r",capture1, capture2, capture3);
    printf("flag = %u (if = 1, TMR1 overflowed, timing will be off)\n\r",flag);
    T_ON = (capture2 - capture1)*1.0*4.0/_XTAL_FREQ;  // convert to seconds
    T_PWM = (capture3 - capture1)*1.0*4.0/_XTAL_FREQ;  // convert to seconds
    printf("T_ON = %f s, T_PWM = %f s\n\r",T_ON, T_PWM);
    IO_RD2_SetLow(); // LED off, capture finished

    while (1);
}
```

# Chapter 13. Signal Measurement Timer (SMT)

The Signal Measurement Timer (SMT) is a newer peripheral available from Microchip. It is a multifunction peripheral that combines timer and capture features into a powerful package that is often more useful than the timer and capture modules we have already looked at. Part of the power comes from having 24-bit memory buffers allowing longer time intervals to be set or measured. For capture, SMT features two buffers, one for rising edges and one for falling edges. It also has a nice feature that certain modes can reset the timer to zero automatically at the first edge. With the Capture modules in the previous chapter, you manually set the timer to zero and waited for the next edge. If that took a lot of time, the timer could roll over on one the next captures. This was inefficient. The two buffers are called the Capture Period (SMT1CPR) and Capture Pulse Width (SMT1CPW) registers.

**Warning**: Use unsigned long variables with SMT SFRs and functions.

## Timer Functions

A timer is a peripheral that monitors an internal clock. An internal clock is just a continuous square wave signal of constant frequency $f$. The SMT1 timer has two parts; a *period* which is a number between 0 and $2^{24} - 1$ and a *counter* (also called the *timer*) which increments automatically on every $N^{th}$ rising edge of the clock signal. The value $N$ is called the *prescaler*. When the counter reaches the value of the period, at the next increment it rolls over to zero. In the process, it sets a flag to have value one. The counter can be read at any instant you choose, say before a chunk of code or when a button is pressed. It can also be read after that chunk of code or when the button is released. The time interval is then

$$time = [final\_count - initial\_count] \times N \times \frac{1}{f}. \quad [13\text{-}1]$$

You do need to check that the counter has not rolled over between those two readings for the above formula to be correct. You check by making sure that the flag is not set after the second reading.

Timers are also used to create specific delays. The period is chosen to give the specific delay or time interval needed. The flag and counter are *reset* or *cleared* or set to zero and you wait until the flag is *set* or changed to 1 when the counter rolls over. The time interval in this case is

$$time = period \times N \times \frac{1}{f}. \quad [13\text{-}2]$$

The MPLAB Code Configurator (MCC) plug-in lets you configure the SMT peripheral as a timer in whichever fashion you wish to use it. In Easy Setup, check the *Enable SMT* and *Enable Data Acquisition After Init* boxes and set *Operation Mode* to Timer as shown in Figure 13-1. Under *Clock Select*, you can choose either HFINTOSC 16 MHz, FOSC, or FOSC/4 as the frequency to

use. The *Prescaler* uses every $N^{th}$ edge of the timing signal. The possible prescaler values are 1, 2, 4, or 8. Set the *Period* to a number from 0 to $2^{24} - 1 = 16{,}777{,}215$. You do not have to enter the value in hexadecimal, a decimal number will work just as well.

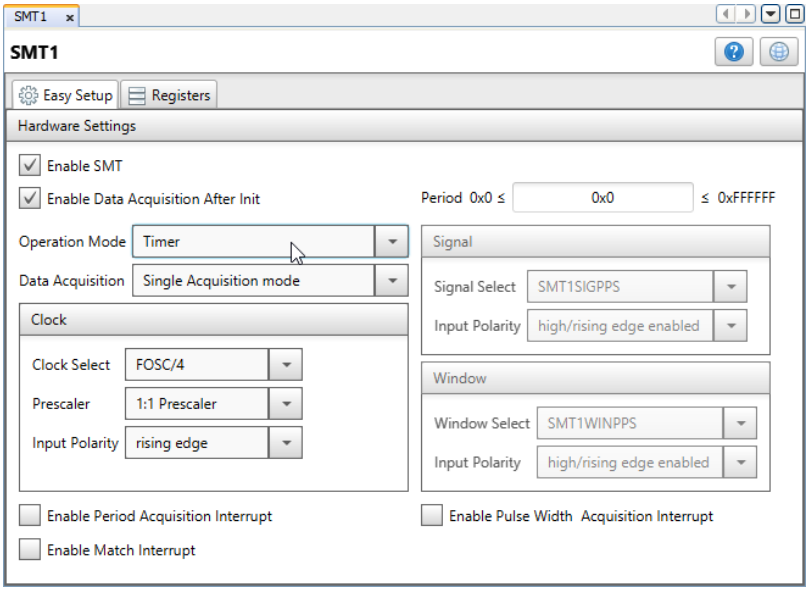



**Figure 13-1:** SMT configured as a timer.

When the SMT library files, smt1.c and smt1.h, are generated by MCC, they will contain a number of helpful functions. Explanations of the functions and SFRs you can use are given in Table 13-1.

**Table 13-1:** SMT1 Functions and SFRs.

| Function and SFRs | Explanation |
|---|---|
| uint32_t SMT1_GetPeriod(void) | Reads the period, a number between 0 and $2^{24} - 1$. Flag SMT1IF is set when counter reaches this value. |
| void SMT1_SetPeriod(uint32_t periodVal) | Write the period. Must be a number between 0 and $2^{24} - 1$. Flag SMT1IF is set when counter reaches this value. |
| void SMT1_ManualTimerReset(void) | Resets the SMT1 counter to zero. |
| uint32_t SMT1_GetTimerValue(void) | Read the current value of the SMT1 counter. |
| void SMT1_DataAcquisitionEnable(void) | Start the counter incrementing. |
| void SMT1_DataAcquisitionDisable(void) | Stop the counter from incrementing. |
| bool SMT1_IsTimerIncrementing(void) | Checks if timer is incrementing. |
| void SMT1_HaltCounter(void) | When counter reaches period, remains at period. Does not roll over. Flag still sets. |
| void SMT1_SingleDataAcquisition(void) | Capture one set of data. Timer stops incrementing after capture completed. |
| void SMT1_RepeatDataAcquisition(void) | Capture data continuously. |
| uint32_t SMT1_GetCapturedPulseWidth() | Reads the CPW buffer. In capture mode, this holds falling edge time. |

| | |
|---|---|
| uint32_t SMT1_GetCapturedPeriod() | Reads the CPR buffer. In capture mode, this holds rising edge time. |
| SMT1IF | Timer flag. Equals 1 when the counter reaches period. |
| SMT1PRAIF | PeRiod Acquisition Interrupt Flag. Equals 1 when rising edge detected. |
| SMT1PWAIF | Pulse Width Acquisition Interrupt Flag. Equals 1 when falling edge detected. |
| SMT1CON1bits.MODE | Read/Write SMT1 Modes<br>Value Mode<br>0 Timer<br>1 Gated Timer<br>2 Period and Duty-Cycle Acquisition<br>3 High and low time measurement<br>6 Time of flight<br>7 Capture<br>8 Counter |
| SMT1CON0bits.PS | Read/Write SMT1 Prescaler value<br>Value N<br>0 1<br>1 2<br>2 4<br>3 8 |
| SMT1CON0bits.EN | Read/Write SMT1 enable status<br>0 = off, 1 - on |

Sample code is provided below to show how to use SMT as a timer, either to time code or to create a non-blocking delay. As a first step, the code prints the initial settings of the mode, period, and prescaler and check that the counter is incrementing. The second step in the code resets mode, period, and prescaler and starts the timer. This is not necessary since it was configured in Easy Setup. The code is merely here to demonstrate how to make change in a program. The next portion uses SMT to time a delay. The time is reset to zero before the delay. The last section demonstrates a non-blocking delay. Note that you should clear the flag and reset the counter before waiting for the counter to roll over and the flag to set.

```
// SMT-Timer.c
// FOSC at 4 MHz, External Oscillator for accuracy
// UART at 9600 bps
// LED on RD2
//
// SMT1 using FOSC/4, Prescaler = 1, period = 16,000,000 us = 16 seconds
// WARNING! period must be bigger than expected signal time.
// Timer Mode, Timing code execution & non-blocking delay

#include "mcc_generated_files/mcc.h"
#include "putty.h"
#include <stdbool.h>
#include <stdio.h>     // has printf()
```

```
void main(void)
{
    // Initialize the device
    SYSTEM_Initialize();

    uint32_t time1=1, time2;
    float time;
    unsigned char letter;

    IO_RD2_SetHigh(); // LED on show capture in progress

    clearPuTTY();

    printf("SMT1 Timer\n\n\r");

    printf("SMT Mode = %u (0 is timer)\n\r",SMT1CON1bits.MODE);
    printf("SMT enabled = %u (0 is no, 1 is yes)\n\r",SMT1CON0bits.EN);
    printf("SMT timer incrementing = %u (0 is no,
                    1 is yes)\n\r",SMT1_IsTimerIncrementing());
    printf("SMT prescaler setting = %u , N = %u \n\r",SMT1CON0bits.PS,
                    1 << SMT1CON0bits.PS);
    printf("SMT period %lu \n\n\r",SMT1_GetPeriod());

    //codes to control operation. Not needed, set in MCC Easy Setup
    SMT1CON0bits.EN = 1;          // enables SMT peripheral
    SMT1_DataAcquisitionEnable(); // start SMT peripheral
    SMT1_SetPeriod(16000000ul);   // set period
    SMT1CON0bits.PS = 0;          // set prescaler N = 2^0 = 1

    SMT1_ManualTimerReset();       // zero SMT1 counter/timer
    time_1 = SMT1_GetTimerValue(); // just to check if zeroed
    DELAY_milliseconds(3100);
    time_2 = SMT1_GetTimerValue();
    printf("SMT start = %lu musec (should be very small number)\n\r", time_1);
    printf("SMT end = %lu musec\n\r", time_2);
    time = (float)(time_2 - time_1) * (1 << SMT1CON0bits.PS) * 4.0/_XTAL_FREQ;
    printf("elapsed time = %f seconds\n\n\r", time);

    // 16 second non-blocking delay
    printf("16 s non-blocking delay - type letters please \n\r");

    SMT1IF = 0;  // clear SMT timer interrupt
    SMT1_ManualTimerReset();
    while(!SMT1IF)
    {
      if(UART2_DataReady)
      {
        letter = UART2_Read();    // read a single character
        printf(" You have typed {%c}, ascii #%u \n\r",letter, letter);
      }
    }
    printf("\nNon-blocking delay finished\n\r");
    IO_RD2_SetLow();         // LED off, timing finished
    while (1);
}
```

## SMT Counter

In Counter mode, the SMT1 peripheral counts rising edges on a pin instead of on an internal clock. The MPLAB Code Configurator (MCC) plug-in lets you configure the SMT peripheral as a counter. In Easy Setup, check the *Enable SMT* and *Enable Data Acquisition After Init* boxes and set *Operation Mode* to Counter as shown in Figure 13-2. Set the *Period* to a number from 0 to $2^{24} - 1 = 16{,}777{,}215$. Period should be a number greater than the number of edges you wish to count. You do not want the counter to roll over in the middle of counting. You do not have to enter the value in hexadecimal, a decimal number will work just as well.

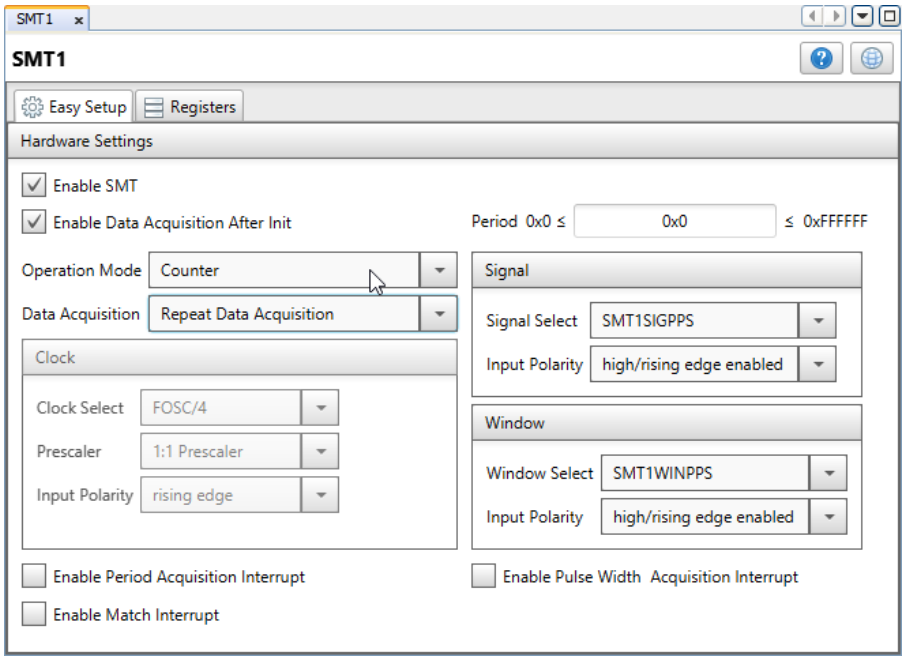



**Figure 13-2:** Configuring SMT as a counter.

The external signal whose edges you are counting must be connected to the SMT1SIG pin. Use Pin Manager: Grid View to choose the pin that will be SMT1SIG as shown in Figure 13-3.

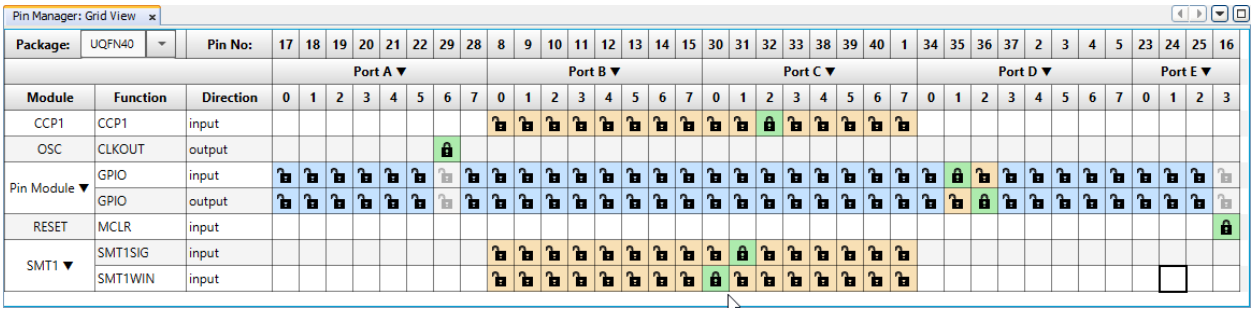



**Figure 13-3:** Designating the SMT1SIG pin.

In your program use the function `SMT1_GetTimerValue()` to check how many edges have been counted. It is probably a good idea to zero the counter using the function

`SMT1_ManualTimerReset()` before the edges start arriving. The following code shows how to use SMT to count button presses.

```
// SMT-Counter.c
// FOSC = 32 MHz
// UART 9600
// SMT counter mode.
// Set pin SMT1SIG to RC1. Attach button to RC1/SMT1SIG

#include "mcc_generated_files/mcc.h"
#include <stdio.h>    // C Library for printf()
#include <stdbool.h>
#include "putty.h"

int main(void)
{
 // Initialize the device
 SYSTEM_Initialize();

 unsigned int newcount = 0, oldcount = 0;

 clearPuTTY();                // clear display
 printf("Test SMT as Counter \n\r");

 printf("Initial settings: Mode = %u (8 is counter), period = %lu,
          initial count = %lu \n\n\r", SMT1CON1bits.MODE,
          SMT1_GetPeriod(), SMT1_GetTimerValue());

 // counter range is 0 to 10
 SMT1_ManualTimerReset();        // reset counter
 printf("Ready \n\n\r");
 while(1){  // only shows 10 presses
     newcount = SMT1_GetTimerValue();
     DELAY_milliseconds(10);
     if (newcount > oldcount){ // print only on change
         if( newcount > (oldcount + 1) )
         {
             newcount = oldcount + 1;
         }
         printf("Count = %u \n\r", newcount);
         oldcount++;
     }

     if (newcount >= 10){
         SMT1_ManualTimerReset(); // reset count to 0
         oldcount = 0;
     }

 }
} // end program
```

## SMT Capture

The capture function in SMT is much like the capture function in the CCP peripherals. The main difference is that CCP capture only has one buffer to hold the time of a rising or a falling edge. It also has only one flag that is set when an edge is detected. The SMT capture function has two buffers, the Capture PeRiod register or SMT1CPR and the Capture Pulse Width or SMT1CPW register, and two flags, the PeRiod Acquisition Interrupt Flag or SMT1PRAIF and the Pulse Width Interrupt Flag or SMT1PWAIF, one of each for rising edges and one of each for falling edges.

A capture peripheral works by watching then external digital signal and noting the times when rising edges and/or falling edges arrive in a buffer which is a temporary memory register. The typical input signals you might want to measure include the duration of a single pulse which might come from a single button press or a flag passing through a photogate, the period and duty cycle of a continuous train of identical square waves which might be timing information from a device, or the on times and off times of a train of arbitrary pulses such as when a picket fence falls through a photogate. Plots of such signals are shown in Figure 13-4 below.

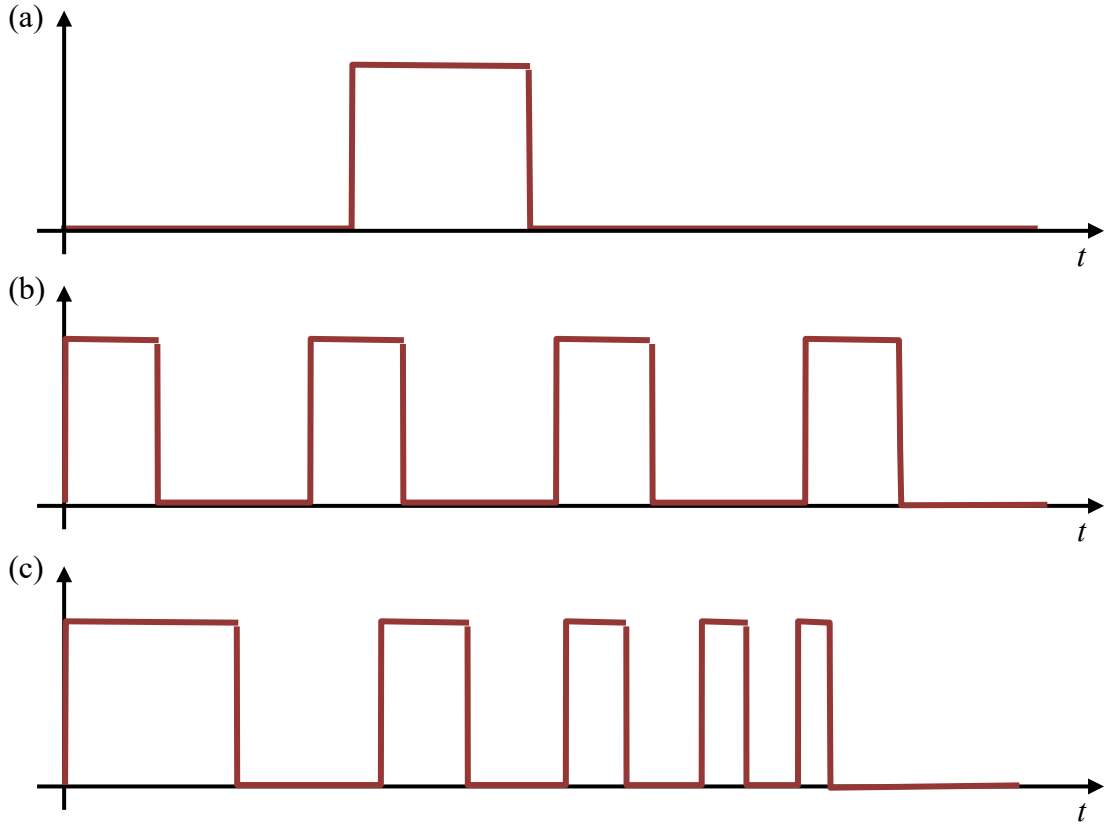



**Figure 13-4:** Typical digital input: (a) pulse, (c) continuous square wave, and (c) arbitrary.

For the single pulse, one need only capture a rising edge and a subsequent falling edge to time the pulse width. For the continuous square wave, one needs at least three consecutive edges to determine pulse width and period. A cautious person would capture more and get several pulse width and period measures to check for consistency. For the arbitrary signal, one would most likely know the number of edges that will occur and would want to capture them all.

Capture works by looking for the edges in an input signal. When a rising edge occurs, the module reads the associated timer module counter and writes that value a rising edge buffer. It also sets the rising edge capture or acquisition flag to let you know rising edge data is available. Similarly, when a falling edge occurs, the module reads the timer and writes the value to a falling edge buffer. A falling edge capture flag is set to let you know falling edge data is available. It is important to read from the buffers before the next similar edge occurs because the new value will overwrite the old value. The behaviour of the flags and buffers is illustrated in Figure 13-6 below. You must clear the capture flags in your code before starting capture and after recording the buffer data.

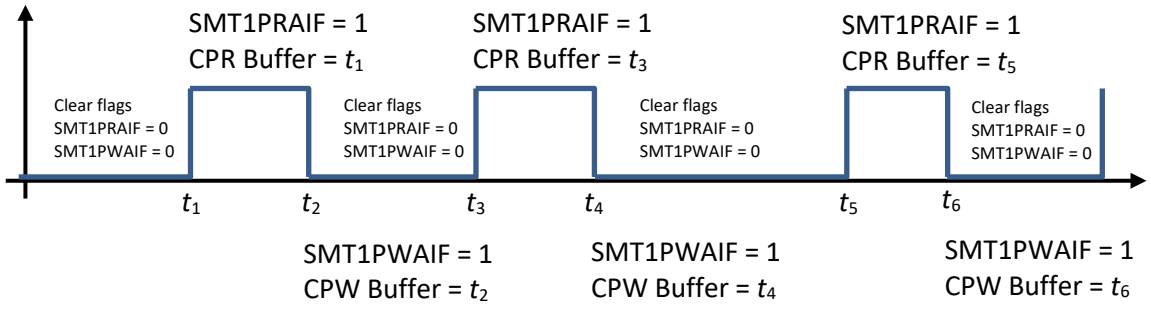



**Figure 13-5:** SMT capture behaviour.

Note that the times, $t_1$ to $t_6$, in Figure 13-6 above are not really times but counts of the number of edges of the internal clock square-wave signal. They can be converted to time using

$$time = t_n \times N \times \frac{1}{f} \qquad [13\text{-}3]$$

where $N$ is the timer prescaler and $f$ is the frequency of the timer.

To keep the values of the timer small, it is a good idea to zero the timer counter and timer flag SMT1IF before measurements start. When saving the buffer data, it is also a good idea to check the value of SMT1IF to ensure that the timer has not rolled over between measurements.

The MPLAB Code Configurator (MCC) plug-in lets you configure the SMT peripheral for capture. In Easy Setup, check the *Enable SMT* and *Enable Data Acquisition After Init* boxes and set *Operation Mode* to Capture as shown in Figure 13-6. Also set *Data Acquisition* to *Repeat Data Acquisition*. Under *Clock Select*, you can choose either HFINTOSC 16 MHz, FOSC, or FOSC/4 as the timing frequency to use. The *Prescaler* uses every $N^{th}$ edge of the timing signal. The possible prescaler values are 1, 2, 4, or 8. Set the *Period* to a number from 0 to $2^{24} - 1 = 16{,}777{,}215$. You do not have to enter the value in hexadecimal, a decimal number will work just as well. The active block *Window* indicates that the signal must be input at the SMT1WIN pin which you should designate in Pin Manager: Grid View. In the same block, under *Input Polarity* you can indicate if you want the capture to begin on a rising edge or a falling edge.

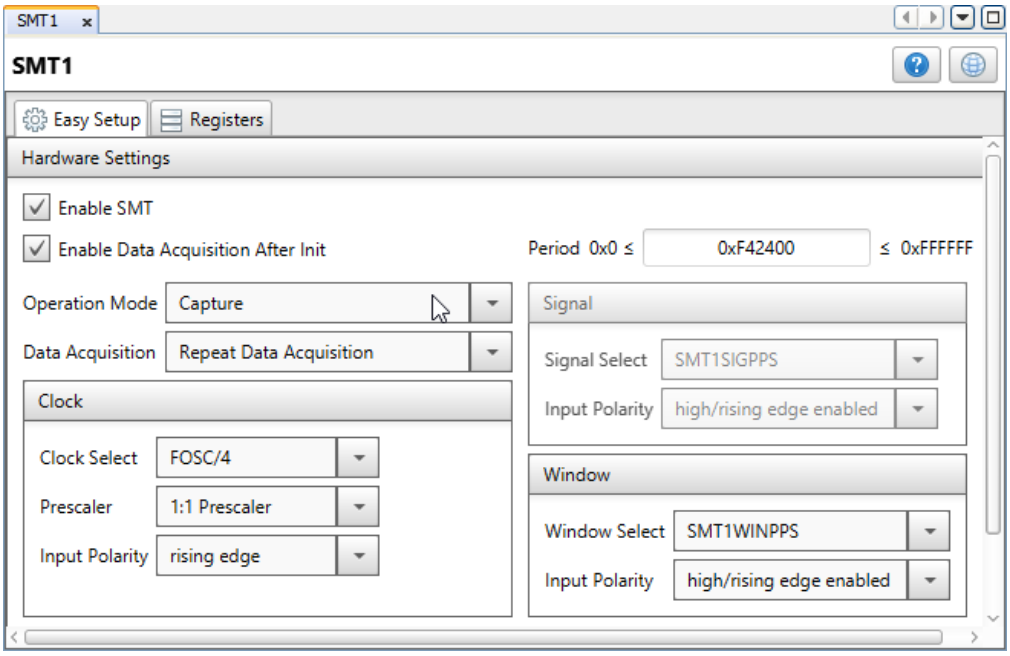



**Figure 13-6:** Configuring SMT for capture.

The MCC generated library, smt1.h and smt1.c, has functions to let you read the butters. These are `SMT1_GetCapturedPeriod()` for the rising edges and `SMT1_GetCapturedPulseWidth()` for the falling edges. The capture flag for a rising edge is `SMT1PRAIF` and the capture flag for a falling edge is `SMT1PWAIF`. The following example code shows the capture of six edges. The timer is zeroed before data collection starts. The data is from a PWM signal for convenience, so period and pulse width are calculated. The program waits for both a rising edge and a falling edge to have been detected before reading the buffers. That is not necessary, you could have waited for a rising edge, read the buffer and repeated for a falling edge. Also, you might only be interested in rising edges. If so, you do not have to bother with finding and recording the falling edge times. The results are printed out at the end of the program to avoid slow printing from interfering with edge finding.

```
/ SMT-Capture.c
// FOSC at 4 MHz, External Oscillator for accuracy
// UART at 9600 bps
// TMR2 using FOSC/4 at Prescaler = 128, period = 32 ms
// PWM5 at 249*128/4*4/FOSC = 7.968 ms (DC = 25%), output pin is RC7
// PWM5 is signal we wish to capture. Connect RC7 to RC5
// Start button on RD1
// LED on RD2
//
// SMT1 using FOSC/4, Prescaler = 1, period = 16,000,000 us = 16 seconds
// WARNING! period must be bigger than expected signal time.
// Capture Mode, SMT1WIN on RC4, start on rising edge

#include "mcc_generated_files/mcc.h"
#include "putty.h"
#include <stdbool.h>
```

```
void main(void)
{
 // Initialize the device
 SYSTEM_Initialize();

 uint32_t time1=1, time2=2, time3=3, time4=4;
 uint32_t time5=5, time6=6;

 while(IO_RD1_GetValue() == 0); //wait for button press;
 IO_RD2_SetHigh(); // LED on show capture in progress

 clearPuTTY();

 printf("SMT1 Capture Test - SMT1WIN on RC4\n\r");

 printf("SMT Mode = %u \n\n\r",SMT1CON1bits.MODE);

 //codes to control operation. Already set in Easy Setup
 SMT1CON0bits.EN = 1;             // enable SMT peripheral
 SMT1_DataAcquisitionEnable();    // start timer
 SMT1_RepeatDataAcquisition();    // capture continuously

 SMT1_ManualTimerReset();         // zero SMT1 counter/timer

 SMT1PRAIF = 0;          // clear buffer interrupt flags
 SMT1PWAIF = 0;
 while(SMT1PRAIF == 0 || SMT1PWAIF == 0 ); // wait for both buffers to fill
 time1 = SMT1_GetCapturedPeriod();     // holds rising edge time
 time2 = SMT1_GetCapturedPulseWidth(); // holds falling edge time

 SMT1PRAIF = 0;         // clear buffer interrupt flags
 SMT1PWAIF = 0;
 while(!SMT1PRAIF || !SMT1PWAIF );      // wait for both buffers to fill
 time3 = SMT1_GetCapturedPeriod();      // holds rising edge time
 time4 = SMT1_GetCapturedPulseWidth();  // holds falling edge time

 SMT1PRAIF = 0;         // clear buffer interrupt flags
 SMT1PWAIF = 0;
 while(!SMT1PRAIF || !SMT1PWAIF );      // wait for both buffers to fill
 time5 = SMT1_GetCapturedPeriod();      // holds rising edge time
 time6 = SMT1_GetCapturedPulseWidth();  // holds falling edge time

 printf("Captured sequence is RE, FE, RE \n\r");
 printf("time1 = %lu, time2 = %lu, time3 = %lu \n\r", time1, time2, time3);
 printf("time4 = %lu, time5 = %lu, time6 = %lu \n\n\r", time4, time5, time6);
 printf("Period = %lu musec, Pulse width = %lu musec \n\r", time3 - time1,
                                                           time2 - time1);
 printf("Period = %lu musec, Pulse width = %lu musec \n\r", time6 - time4,
                                                           time6 - time5);

 IO_RD2_SetLow(); // LED off, capture finished

 while (1);
}
```

## SMT Specialty Capture Modes

The SMT Capture function described in the previous section is a powerful and general function that lets you time the edges of all sorts of incoming square wave signals. Sometimes the incoming signals are simpler. SMT has a neat trick of starting the timer at zero when it locates the first edge in certain modes. As a result, the duration of features such as pulse width, period, and off time can be stored in the buffers. This means less calculation need to be done. It also means that the timer is far less likely to roll over between calculations.

The MPLAB Code Configurator (MCC) plug-in lets you configure the SMT peripheral for these capture modes as usual. In Easy Setup, check the *Enable SMT* and *Enable Data Acquisition After Init* boxes and set *Operation Mode* to the particular mode desired as shown in Figure 13-7. Also set *Data Acquisition* to *Repeat Data Acquisition*. Under *Clock Select*, you can choose either HFINTOSC 16 MHz, FOSC, or FOSC/4 as the timing frequency to use. The *Prescaler* uses every $N^{th}$ edge of the timing signal. The possible prescaler values are 1, 2, 4, or 8. Set the *Period* to a number from 0 to $2^{24} - 1 = 16{,}777{,}215$. You do not have to enter the value in hexadecimal, a decimal number will work just as well.

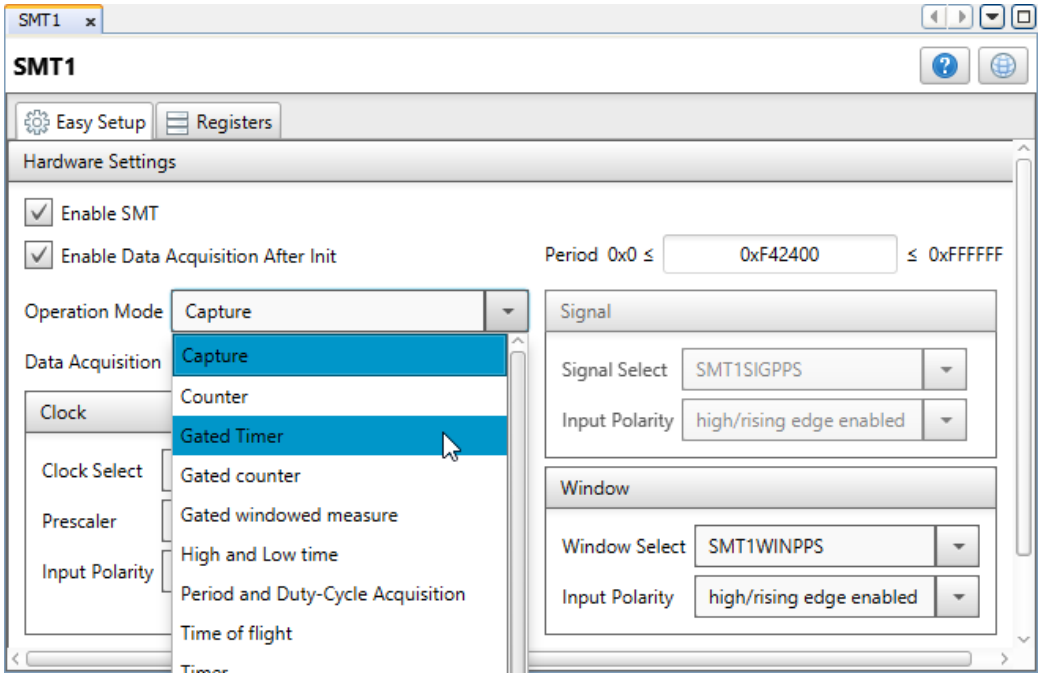



**Figure 13-7:** Choosing the specialty capture modes.

When an Operation Mode is chosen, the blocks labelled Signal and Window on the left side of Figure 13-7 may be grayed out or active. These refer to the SMT1SIG and SMT1WIN input pins. Generally, the block that is active is the pin you must configure in the Pin Manager: Grid View tab and to which you connect your input signal. In the blocks, under *Input Polarity* you can indicate if you want the capture to begin on a rising edge or a falling edge. Note that all the descriptions in this chapter assume that you start on a rising edge. The mode and the input pin are summarized in Table 13-2 below.

**Table 13-2:** SMT1 mode & input pin

| SMT1CON1bits.MODE | Mode | Input Pin |
|---|---|---|
| 0 | Timer | None |
| 1 | Gated Timer | SMT1SIG |
| 2 | Period and Duty-Cycle Acquisition | SMT1SIG |
| 3 | High and low time measurement | SMT1SIG |
| 6 | Time of flight | SMT1WIN & SMT1SIG |
| 7 | Capture | SMT1WIN |
| 8 | Counter | SMT1SIG |

## Gate Timer Mode

Many transduces send their measurement as a pulse. The duration of the pulse is proportional to the strength of the effect. Gate Timer Mode is ideal for measuring pulse widths. In Gate Timer Mode, the timer does not turn on until the first rising edge, so the first rising edge is at zero. This is shown in Figure 13-8. When the first falling edge is encountered, the value of the time is added to the SMT1CPW buffer. By definition, this is the pulse width, or on time, or duty cycle of the signal. The falling edge flag, SMT1PWAIF, is set as well. You monitor this flag to know when the operation has completed and to know when to record the buffer value. You then need to clear the flag in your code for the next measurement. At the falling edge, the timer stops incrementing. At the next rising edge, the timer starts incrementing again. This stoppage is noted by the purple arrows in Figure 13-8. When the second falling edge is encountered, the timer value is added to the SMT1CPW buffer. The previous value in the buffer is not overwritten and replaced. Instead, the buffer now holds the sum of the two pulse widths. This will continue for however many pulses are received. The falling edge flag, SMT1PWAIF, is set at every falling edge.

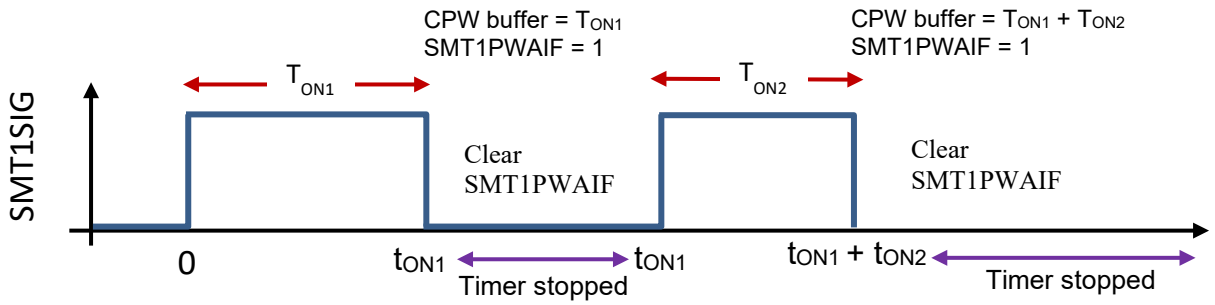



**Figure 13-8:** Gate Timer Mode behaviour.

If you do not wish to have the cumulative pulse width in the buffer, just the individual pulse widths, you would need to reset the timer to zero after reading the buffer using function `SMT1_ManualTimerReset()`. The behaviour would look like Figure 13-9. The change is subtle; note that each rising edge is labelled time 0. The example code below shows the steps to collect three pulse widths. For each measurement, the timer and flag are reset to zero. The program then waits until the flag is set. The buffer is read with the function `SMT1_GetCapturedPulseWidth()`. Please note that you can collect the cumulative pulse width times by commenting out the second

and third call to `SMT1_ManualTimerReset()`. The results are printed out at the end of the program to avoid slow printing from interfering with edge finding.

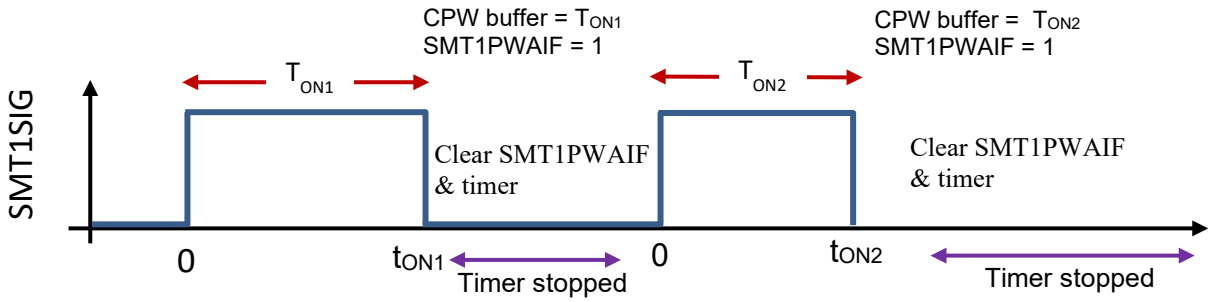



**Figure 13-9:** Gate Timer Mode behaviour with timer reset.

```
// SMT-GatedTime.c
// FOSC at 4 MHz, External Oscillator for accuracy
// UART at 9600 bps
// TMR2 using FOSC/4 at Prescaler = 128, period = 32 ms
// PWM5 at 249*128/4*4/FOSC = 7.968 ms (DC = 25%), output pin is RC4
// PWM5 is signal we wish to capture. Connect to RC5
//
// SMT1 using FOSC/4, Prescaler = 1, period = 16,000,000 us = 16 seconds
// WARNING! period must be bigger than expected signal time.
// Gated Timer Mode, RE to FE on SMT1SIG (RC5)  Pulse Width
//
// Reads 3 pulse widths

#include "mcc_generated_files/mcc.h"
#include "putty.h"
#include <stdbool.h>

void main(void)
{
    // Initialize the device
    SYSTEM_Initialize();

    uint32_t time1=1, time2=2, time3=3;

    clearPuTTY();

    printf("SMT1 Gated Timer - RE to FE on SMT1SIG (RC5)\n\n\r");

    printf("SMT Mode = %u \n\n\r",SMT1CON1bits.MODE);

    //codes to control operation. Not needed. Already set in Easy Setup
    SMT1CON0bits.EN = 1;             // enable SMT peripheral
    SMT1_DataAcquisitionEnable();    // allows acquisitions/captures
    SMT1_RepeatDataAcquisition();    // allow repeated measurements

    // capture 3 pulse widths
    SMT1_ManualTimerReset();  // zero SMT1 counter/timer
    SMT1PWAIF = 0;            // clear buffer interrupt flags
    while(SMT1PWAIF == 0);    // wait for pulse width buffer to fill
    time1 = SMT1_GetCapturedPulseWidth();  // read buffer
```

```
    SMT1_ManualTimerReset();  // zero SMT1 counter/timer
                              // remove for cumulative timing
    SMT1PWAIF = 0;            // clear buffer interrupt flags
    while(SMT1PWAIF == 0);    // wait for pulse width buffer to fill
    time2 = SMT1_GetCapturedPulseWidth();  // read buffer

    SMT1_ManualTimerReset();  // zero SMT1 counter/timer
                              // remove for cumulative timing
    SMT1PWAIF = 0;            // clear buffer interrupt flags
    while(SMT1PWAIF == 0);    // wait for pulse width buffer to fill
    time3 = SMT1_GetCapturedPulseWidth();

    printf("Pulse width = %lu musec\n\r", time1);
    printf("Pulse width = %lu musec\n\r", time2);
    printf("Pulse width = %lu musec\n\r", time3);

    while (1);
}
```

## Period and Duty Cycle Acquisition Mode

We will wish to measure the period and the pulse width of continuous square wave signals. The Period and Duty Cycle Acquisition Mode is ideal for this purpose. In this mode, see Figure 13-10, the timer starts to increment on the first rising edge. At the first falling edge, the timer value is written to the Capture Pulse Width (CPW) buffer and the Pulse Width Acquisition Interrupt Flag, SMT1PWAIF, is set to 1. The CPW buffer holds the pulse width. The timer keeps incrementing and at the next rising edge, the timer value is written to the Capture PeRiod (CPR) buffer and the PeRiod Acquisition Interrupt Flag, SMT1PRAIF, is set to 1. The CPR buffer holds the period. The timer also resets to zero to allow measurement of the next pulse width in the train if the mode is running the repeat acquisition setting. The buffers should be read at this point and the flags should be cleared.

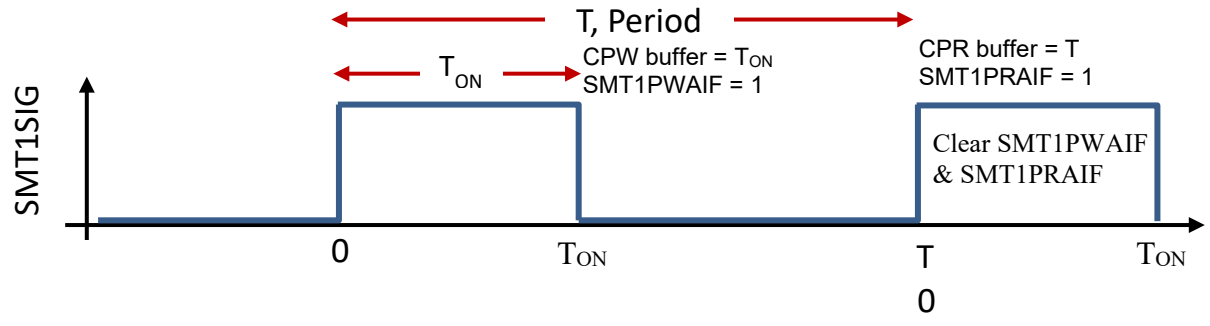



**Figure 13-10:** Pulse width and Period behaviour.

An example program below demonstrates the code to get three measurements of a PWM signal. Each time, the flags are cleared and the program waits for both flags to be set to 1, that is until both edges are found. At that point, the buffers are saved to memory with the functions

`SMT1_GetCapturedPulseWidth()` and `SMT1_GetCapturedPeriod()`. The results are printed out at the end of the program to avoid slow printing from interfering with edge finding.

```
// SMT-PeriodAndPulse.c
// FOSC at 4 MHz, External Oscillator for accuracy
// UART at 9600 bps
// TMR2 using FOSC/4 at Prescaler = 128, period = 32 ms
// PWM5 at 249*128/4*4/FOSC = 7.968 ms (DC = 25%), output pin is RA7
// PWM5 is signal we wish to capture
// Start button on RD1
// LED on RD2
//
// SMT1 using FOSC/4, Prescaler = 1, period = 16,000,000 us = 16 seconds
// WARNING! period must be bigger than expected signal time.
// Period & Duty Cycle Mode, SMT1SIG on RC5, start on rising edge
//
// Measure period and pulse width three times

#include "mcc_generated_files/mcc.h"
#include "putty.h"
#include <stdbool.h>

void main(void)
{
    // Initialize the device
    SYSTEM_Initialize();

    uint32_t time1=1, time2=2, time3=3, time4=4;
    uint32_t time5=5, time6=6;

    while(IO_RD1_GetValue() == 0); //wait for button press;
    IO_RD2_SetHigh(); // LED on show capture in progress

    clearPuTTY();

    printf("SMT1 Period and Duty cycle Test - SMT1SIG on RC1\n\r");

    printf("SMT Mode = %u \n\n\r",SMT1CON1bits.MODE);

    //codes to control operation. Not needed. Already set in Easy Setup
    SMT1CON0bits.EN = 1;             // enable SMT peripheral
    SMT1_DataAcquisitionEnable();    // allows acquisitions/captures
    SMT1_RepeatDataAcquisition();    // allow repeated measurements

    SMT1PRAIF = 0;          // clear period (RE) interrupt flag
    SMT1PWAIF = 0;          // clear pulse width (FE) interrupt flag
    while(SMT1PRAIF == 0 || SMT1PWAIF == 0 ); // wait for buffers to fill
    time1 = SMT1_GetCapturedPeriod();
    time2 = SMT1_GetCapturedPulseWidth();

    SMT1PRAIF = 0;          // clear period (RE) interrupt flag
    SMT1PWAIF = 0;          // clear pulse width (FE) interrupt flag
    while(!SMT1PRAIF || !SMT1PWAIF );  // wait for both buffers to fill
    time3 = SMT1_GetCapturedPeriod();
    time4 = SMT1_GetCapturedPulseWidth();
```

```
    SMT1PRAIF = 0;          // clear period (RE) interrupt flag
    SMT1PWAIF = 0;          // clear pulse width (FE) interrupt flag
    while(!SMT1PRAIF || !SMT1PWAIF );  // wait for both buffers to fill
    time5 = SMT1_GetCapturedPeriod();
    time6 = SMT1_GetCapturedPulseWidth();

    printf("Period = %lu musec, Pulse width = %lu musec \n\r", time1, time2);
    printf("Period = %lu musec, Pulse width = %lu musec \n\r", time3, time4);
    printf("Period = %lu musec, Pulse width = %lu musec \n\r", time5, time6);

    IO_RD2_SetLow(); // LED off, capture finished

    while (1)
    {
        // Add your application code
    }
}
```

## High and Low Time Mode

The High and Low Time Mode is nearly identical to The Period and Duty Cycle Acquisition Mode except that it collects the pulse width duration and off time rather than the pulse width and period. In this mode, Figure 13-11the timer starts to increment on the first rising edge. At the first falling edge, the timer value is written to the Capture Pulse Width (CPW) buffer and the Pulse Width Acquisition Interrupt Flag, SMT1PWAIF, is set to 1. The CPW buffer holds the pulse width. The timer resets to zero, unlike the previous mode, and keeps incrementing. At the next rising edge, the timer value is written to the Capture PeRiod (CPR) buffer and the PeRiod Acquisition Interrupt Flag, SMT1PRAIF, is set to 1. The CPR buffer holds the off time not the period. The timer also resets to zero to allow measurement of the next pulse width in the train if the mode is running the repeat acquisition setting. The buffers should be read at this point and the flags should be cleared.

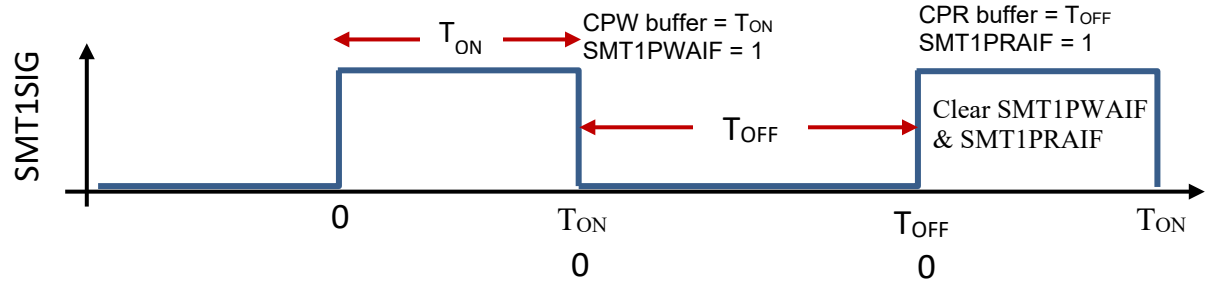



**Figure 13-11:** Pulse width and Off Time behaviour.

An example program below demonstrates the code to get three measurements of a PWM signal. Each time, the flags are cleared and the program waits for both flags to be set to 1, that is until both edges are found. At that point, the buffers are saved to memory with the functions `SMT1_GetCapturedPulseWidth()` and `SMT1_GetCapturedPeriod()`. The results are printed out at the end of the program to avoid slow printing from interfering with edge finding.

```
// SMT-HighAndLow.c
// FOSC at 4 MHz, External Oscillator for accuracy
// UART at 9600 bps
// TMR2 using FOSC/4 at Prescaler = 128, period = 32 ms
// PWM5 at 249*128/4*4/FOSC = 7.968 ms (DC = 25%), output pin is RA7
// PWM5 is signal we wish to capture. Connect to RC5
// Start button on RD1
// LED on RD2
//
// SMT1 using FOSC/4, Prescaler = 1, period = 16,000,000 us = 16 seconds
// WARNING! period must be bigger than expected signal time.
// High and Low Time Mode, SMT1SIG on RC5, start on rising edge
//
// Measure period and pulse width three times

#include "mcc_generated_files/mcc.h"
#include "putty.h"
#include <stdbool.h>

void main(void)
{
  // Initialize the device
  SYSTEM_Initialize();

  uint32_t time1=1, time2=2, time3=3, time4=4;
  uint32_t time5=5, time6=6;

  while(IO_RD1_GetValue() == 0); //wait for button press;
  IO_RD2_SetHigh(); // LED on show capture in progress

  clearPuTTY();

  printf("SMT1 High and Low Time Test - SMT1SIG on RC5\n\r");

  printf("SMT Mode = %u \n\n\r",SMT1CON1bits.MODE);

  //codes to control operation. Not needed. Already set in Easy Setup
  SMT1CON0bits.EN = 1;             // enable SMT peripheral
  SMT1_DataAcquisitionEnable();    // allows acquisitions/captures
  SMT1_RepeatDataAcquisition();    // allow repeated measurements

  SMT1PRAIF = 0;          // clear period (RE) interrupt flag
  SMT1PWAIF = 0;          // clear pulse width (FE) interrupt flag
  while(SMT1PRAIF == 0 || SMT1PWAIF == 0 ); // wait for both buffers to fill
  time2 = SMT1_GetCapturedPulseWidth(); // RE to FE - high time
  time1 = SMT1_GetCapturedPeriod();     // FE to RE - low time, not period

  SMT1PRAIF = 0;          // clear period (RE) interrupt flag
  SMT1PWAIF = 0;          // clear pulse width (FE) interrupt flag
  while(!SMT1PRAIF || !SMT1PWAIF );  // wait for both buffers to fill
  time4 = SMT1_GetCapturedPulseWidth();  // RE to FE - high time
  time3 = SMT1_GetCapturedPeriod();      // FE to RE - low time, not period

  SMT1PRAIF = 0;          // clear period (RE) interrupt flag
  SMT1PWAIF = 0;          // clear pulse width (FE) interrupt flag
  while(!SMT1PRAIF || !SMT1PWAIF );  // wait for both buffers to fill
  time6 = SMT1_GetCapturedPulseWidth();  // RE to FE - high time
```

```
    time5 = SMT1_GetCapturedPeriod();      // FE to RE - low time, not period

    printf("High = %lu musec, Low = %lu musec \n\r", time2, time1);
    printf("High = %lu musec, Low = %lu musec \n\r", time4, time3);
    printf("High = %lu musec, Low = %lu musec \n\r", time6, time5);

    IO_RD2_SetLow(); // LED off, capture finished

    while (1);
}
```

## Time of Flight Mode

This mode differs from the others in having two input pins. The goal of this mode is to capture the time between the first rising edge on the SMT1WIN pin and the first rising edge on the SMT1SIG pin as shown.in Figure 13-12. When the first rising edge is detected on the SMT1WIN pin the timer starts incrementing from zero. When the first rising edge is detected on the SMT1SIG pin the timer value is stored in the Capture PeRiod (CPR) buffer and the PeRiod Acquisition Interrupt Flag, SMT1PRAIF, is set to 1. The timer also resets. The buffer should be read at this point and the flag should be cleared.

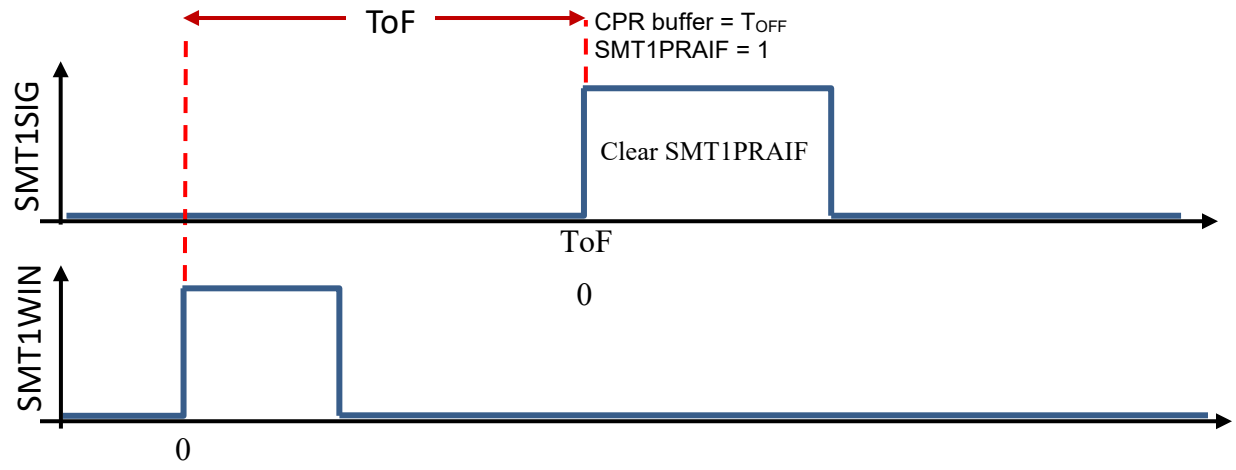



**Figure 13-12:** Time of Flight Mode behaviour.

In the event of a second rising edge occurring on the SMT1WIN pin before there is a first rising edge on the SMT1SIG pin, – a bad measurement – the timer value is stored in the Capture Pulse Width (CPW) buffer and the Pulse Width Acquisition Interrupt Flag, SMT1PWAIF, is set to 1. The behaviour is shown in Figure 13-13. By checking the status of SMT1PWAIF you can determine if there was a bad measurement in this mode.

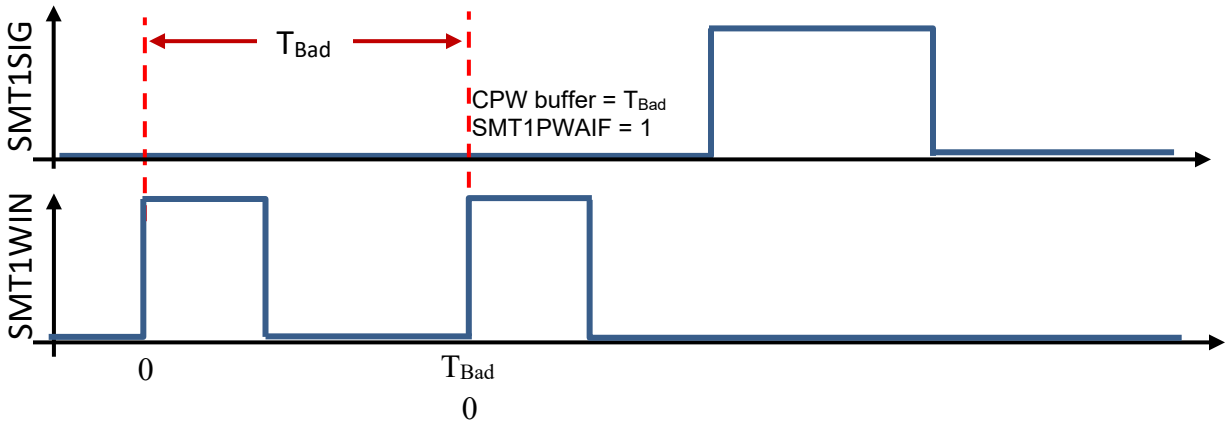



**Figure 13-13:** Time of Flight bad measurement behaviour.

The following example code illustrates the use of the Time of Flight Capture mode. It can be used with two buttons. The project is configured for single data acquisitions (single sets of measurements) but in a continuous loop. This allows for better error checking. The program check to see which flag gets set first. If it is the SMT1PWAIF flag, it reports a bad measurement and disables the acquisition of more edges. An LED turns on to indicate the start of the first button press. It will go off when either button is pressed next.

```
// SMT-TimeOfFlight.c
// FOSC at 4 MHz, External Oscillator for accuracy
// UART at 9600 bps
//
// LED on RD2
//
// SMT1 using FOSC/4, Prescaler = 1, period = 16,000,000 us = 16 seconds
// WARNING! period must be bigger than expected signal time.
//
// Time of flight Mode, RE on SMT1WIN (RC4) to RE on SMT1SIG (RC5)
// Single measurement. Connect RC4 and RC5 together.
// Use to time between different buttons on RC4 and RC5

#include "mcc_generated_files/mcc.h"
#include "putty.h"
#include <stdbool.h>

void main(void)
{
    // Initialize the device
    SYSTEM_Initialize();

    uint32_t time1=1, time2 = 2;

    clearPuTTY();
    IO_RD2_SetLow();          // LED off, capture finished

    printf("SMT1 Time of Flight Test\n\r");

    printf("SMT Mode = %u \n\n\r",SMT1CON1bits.MODE);
```

```
    //codes to control operation. Not needed. Already set in Easy Setup
    SMT1CON0bits.EN = 1;             // enable SMT peripheral
    SMT1_DataAcquisitionEnable();    // allows acquisitions/captures
    SMT1_SingleDataAcquisition();    // allow one set of measurements

    while (1)
    {
    SMT1_DataAcquisitionEnable();// allow next measurement
    SMT1_ManualTimerReset();     // clear timer
    SMT1PRAIF = 0;               // clear interrupt flags
    SMT1PWAIF = 0;
    while(SMT1PRAIF == 0 && SMT1PWAIF == 0) // wait for one buffer to fill
    {
     if(SMT1_GetTimerValue() > 0)
        {
         // timer is incrementing, so first RE on SMT1WIN has been found
         IO_RD2_SetHigh();       // LED on show capture in progress
        }
    }

    if (SMT1PRAIF == 1) // good measurement
        {
         time1 = SMT1_GetCapturedPeriod();   // Time of Flight
         SMT1PWAIF = 0;    // reset
         printf("ToF = %lu musec \n\r", time1);
        }
    if (SMT1PWAIF == 1) // bad measurement
        {
         SMT1_DataAcquisitionDisable(); // don't collect more edges
         printf("Two successive button1 presses - no measurement \n\r");
        }
    IO_RD2_SetLow();         // LED off, capture finished
    }
}
```

# Chapter 14. Analog-to-Digital Conversion (ADC)

The DIO pins work with just two voltage values, low and high, which are typically 0 V and +5 V. As we saw, we need only one bit of memory to store the voltage value as either 0 (low) or 1 (high). Analog to Digital Conversion (ADC or A/D) is a method for measuring an arbitrary voltage signal and storing the value as a binary number. In this chapter, we will explore the simplest method for ADC called the *Method of Successive Approximation* which is qualitatively similar to what happens with the PIC. There are other methods, but we need not concern ourselves with these. In the Method of Successive Approximation, two circuit elements are involved. The first is called a *comparator* and is shown schematically in Figure 13-1. As the name suggests, it compares two voltages and if $V_1$ is greater than or equal to $V_2$ it outputs a 1 (true). If not, it outputs 0 (false).

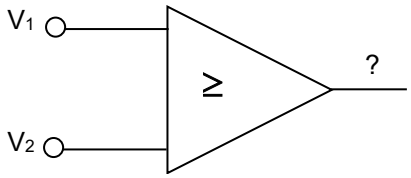



**Figure 14-1:** A comparator.

The second element is a voltage divider with $2^n$ equal resistors connected between reference voltages, $V_{REF+}$ and $V_{REF-}$. Typically, $V_{REF+}$ will be set to $V_{DD}$ which is +5 V and $V_{REF-}$ will be set to $V_{SS}$ which is 0 V. However other reference values may be used. The $2^n$ resistors provide $2^n$ voltage values with which to compare any input voltage. ADC is categorized by the number of comparisons available. The PIC18F46K42 has a 12-bit ADC module so it makes $2^{12} = 4096$ comparisons. As a result, any voltage measurement is stored as a 12-bit binary number. As an aid to understanding the ADC process, a 2-bit example is shown below in Figure 13-2.

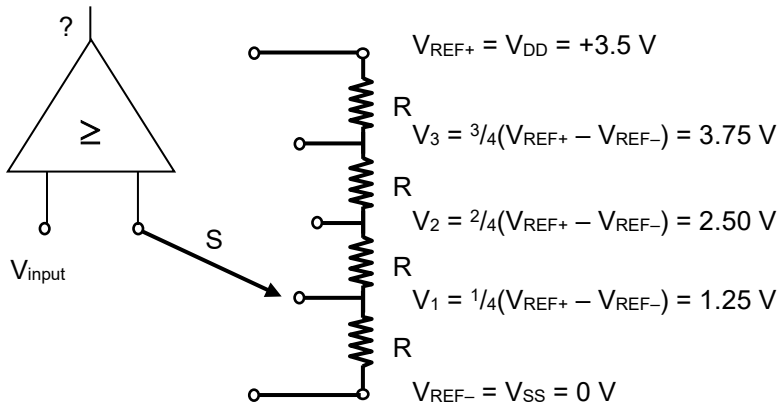



**Figure 14-2:** Comparing an input voltage.

With the $2^2 = 4$ resistors in our voltage divider, we have four voltages we can compare the input signal to: 0 V, 1.25 V, 2.50 V, and 3.75 V. If our input value were 3.05 V for example, this circuit would find that $V_{input}$ lies between 2.50 V and 3.75 V. With 2-bit ADC the result binary or ADC value for the input signal can only be 00, 01, 10, or 11. The binary value for 3.05 V would be 10 as is shown in Figure 13-3. Note that any input voltage between 2.50 V and 3.75 V would be converted to this same ADC value. Thus, this range isn't very precise. However, our MCU actually is 12 bit, not 2 bit, so the precision is much greater.

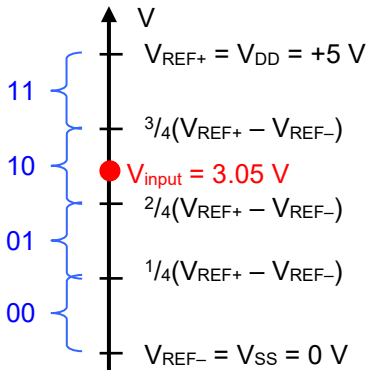



**Figure 14-3:** Binary conversion

The flowchart below, Figure 13-4, indicates how the binary value is created. First you start at the midpoint reference voltage $V_2$ and ask if the input is higher than that value. If yes, the first bit is 1 and you now do a new comparison with $V_3$. Higher than $V_3$ and the last binary digit is 1, else it is 0. If the input voltage was less than $V_2$, the first binary digit is 0. The next comparison is with $V_1$. Higher than $V_1$ and the last binary digit is 1, else it is 0.

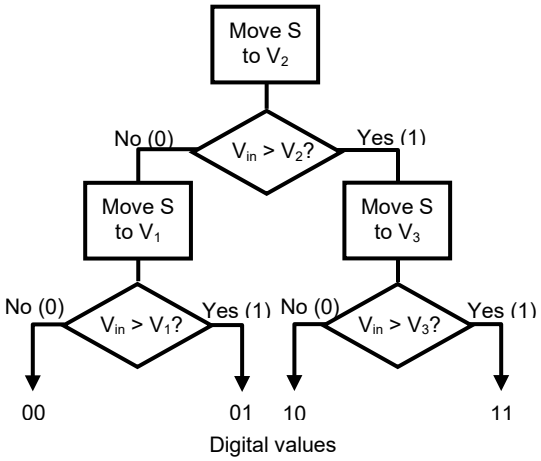



**Figure 14-4:** Comparison flowchart.

Q. If the input voltage was actually $V_{input} = 5.25$ V for the 2-bit example, what would the binary equivalent value be?

Q. If the input voltage was actually $V_{input} = -1.60$ V for the 2-bit example, what would the binary equivalent value be?

Once a signal has been digitized, there is corresponding problem of interpreting the digital value as a voltage. Since each digital value is a bin or range of voltages, it only makes sense to take the midpoint of the range or bin as the reading. For the example, the midpoints are $^1/_8(V_{REF+} - V_{REF-}) = {}^5/_8$ V, $^3/_8(V_{REF+} - V_{REF-}) = {}^{15}/_8$ V, $^5/_8(V_{REF+} - V_{REF-}) = {}^{25}/_8$ V, and $^7/_8(V_{REF+} - V_{REF-}) = {}^{35}/_8$ V. Since the actual reading is anywhere in the range, the precision of the digital measurement is one-half the width of the range, here being $^1/_8(V_{REF+} - V_{REF-}) = {}^5/_8$ V. Thus, four digital values 00, 01,

1,0, and 11 correspond to readings of $^1/_8$V ± $^1/_8$V, 0.375$V_{REF}$ ± 0.125$V_{REF}$, 0.625$V_{REF}$ ± 0.125$V_{REF}$, and 0.875$V_{REF}$ ± 0.125$V_{REF}$. For an n-bit conversion, the general formula is

$$V = \left\{\left(ADCValue + \frac{1}{2}\right) * \frac{[V_{REF+} - V_{REF-}]}{2^n} + V_{REF-}\right\} \pm \frac{1}{2}\frac{[V_{REF+} - V_{REF-}]}{2^n}. \qquad [14\text{-}1]$$

The accuracy of the resulting digital conversion depends on the accuracy of ($V_{REF+} - V_{REF-}$), which may only be accurate to a few percent.

Since the PIC18F46K42 is more adept at handling integers rather than floating type numbers, it is advisable to use $V_{REF}$ in millivolts in the formula above.

## ADC on Time Dependent Input Signals

Varying signals could play havoc with the comparisons we have described above. Each comparison takes time. Suppose the input signal is initially 3.05 V but jumps to 4.15 V at the second comparison. The flowchart will produce a value of 11 instead of 10. Our ADC results will be untrustworthy. To get around this problem, a *sample-and-hold* circuit is used, see Figure 13-7. The signal is connected to a resistor-capacitor circuit via a very fast switch. The resistor and capacitor combination must have a time constant $\tau$ = RC such that the input signal is largely constant over 5 or 10 times that time interval. When ADC is to be performed, the switch is closed long enough for the capacitor to be fully charged and then reopened. The voltage across the capacitor should be very close to the value of the input signal at the time ADC is started and will remain constant if the switch is open. The ADC comparisons are done with respect to the capacitor voltage, not the input signal voltage. The ADC module must have a resistance $R_{ADC}$ much higher than R so that $\tau_{ADC} >> \tau$ and thus $V_C$ does not discharge appreciably during the conversion.

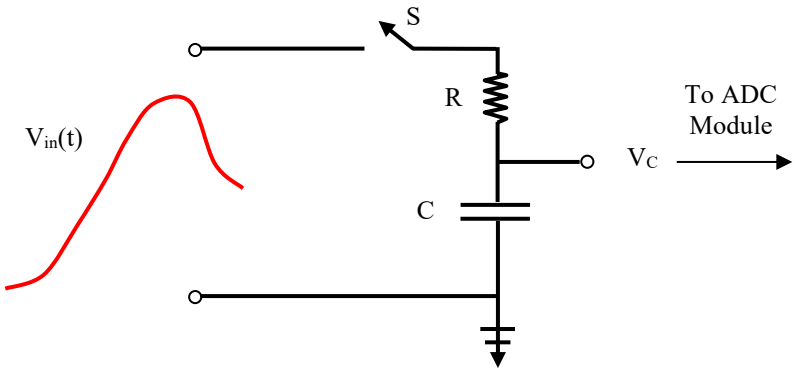



**Figure 14-5:** Sample and hold circuit.

Q. Suppose C = 25 pF and R (actually the series sum of the R, the input resistance, and the switch resistance) is about 4 kΩ. What is $\tau$? How long does it take to fully charge the capacitor (i.e. to more than 99%)?

A second timing factor is that all the ADC comparisons take time to complete. If $T_{AD}$ is the time for a single measurement and conversion, then the variation in the input voltage signal must be

much slower than this. Figure 13-6 shows how the number of ADC bits and size of $T_{AD}$ affect the how well the process captures the input signal. Clearly the more bits involved and the smaller the $T_{AD}$ the better the digitized shape captures the input signal. Note as well that no matter how large N is, the resulting digitization will be poor if the input signal is outside the range set by $V_{REF+}$ and $V_{REF-}$.

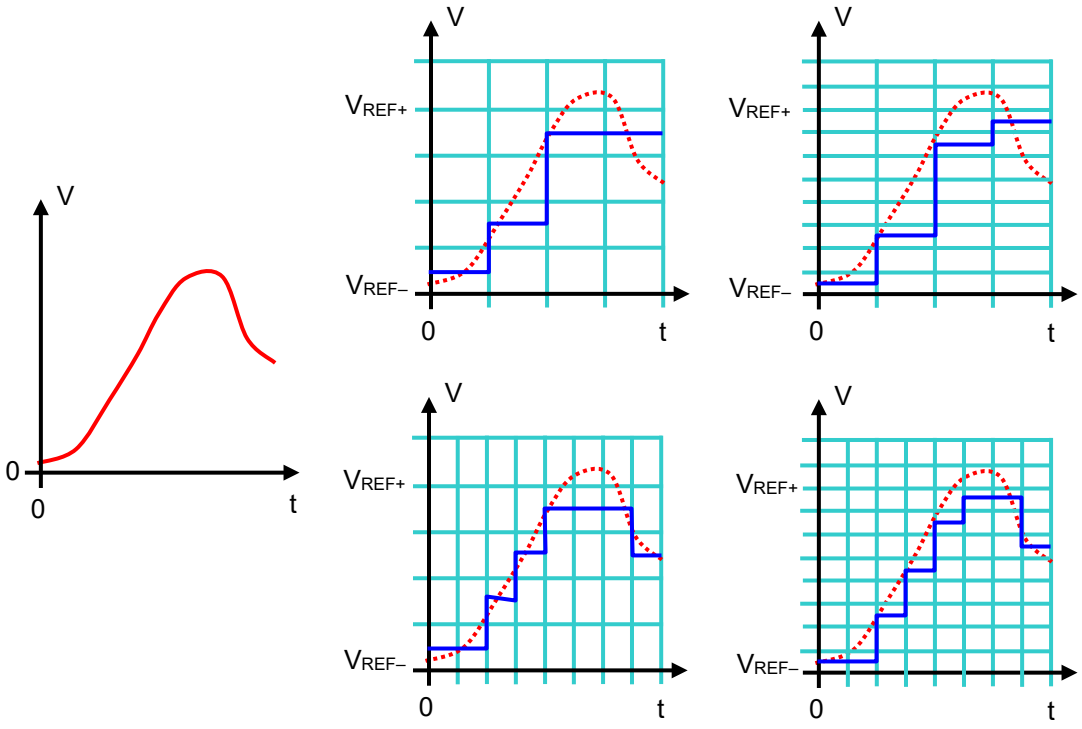



**Figure 14-6:** How N and $T_{ADC}$ affect digitization.

## Configuring ADC with MCC

The MPLAB Code Configurator (MCC) plug-in lets you configure the ADC peripheral. In Easy Setup, shown in Figure 14-8, check the *Enable ADC* box at the top of the tab and choose Basic_mode in the *Operating* dropdown menu. In the *ADC Clock* area, make sure that *Clock Source* is FOSC/ADCLK. The important setting here is *Clock* whereby you ensure that the Sample and Hold circuit and the Successive Approximation circuit have enough time to give accurate results. In the dropdown menu, choose the value of FOSC/N that makes 1 $T_{AD}$ greater than 1 µs. If the setting is too small, there will be a warning in the Notification tab. This area also shows you the maximum *Sampling Frequency* you obtain by taking ADC measurements continuously. It also tells you how much time a single ADC measurement will take, the *Conversion Time*. The sampling frequency and the conversion time are inverses of each other. On the right side, you can set the *Positive Reference*, $V_{REF+}$, and the *Negative reference,* $V_{REF-}$. The simplest choice for these for now is $V_{DD}$ and $V_{SS}$ which are the breadboard voltage and ground. The other choices will be discussed later.

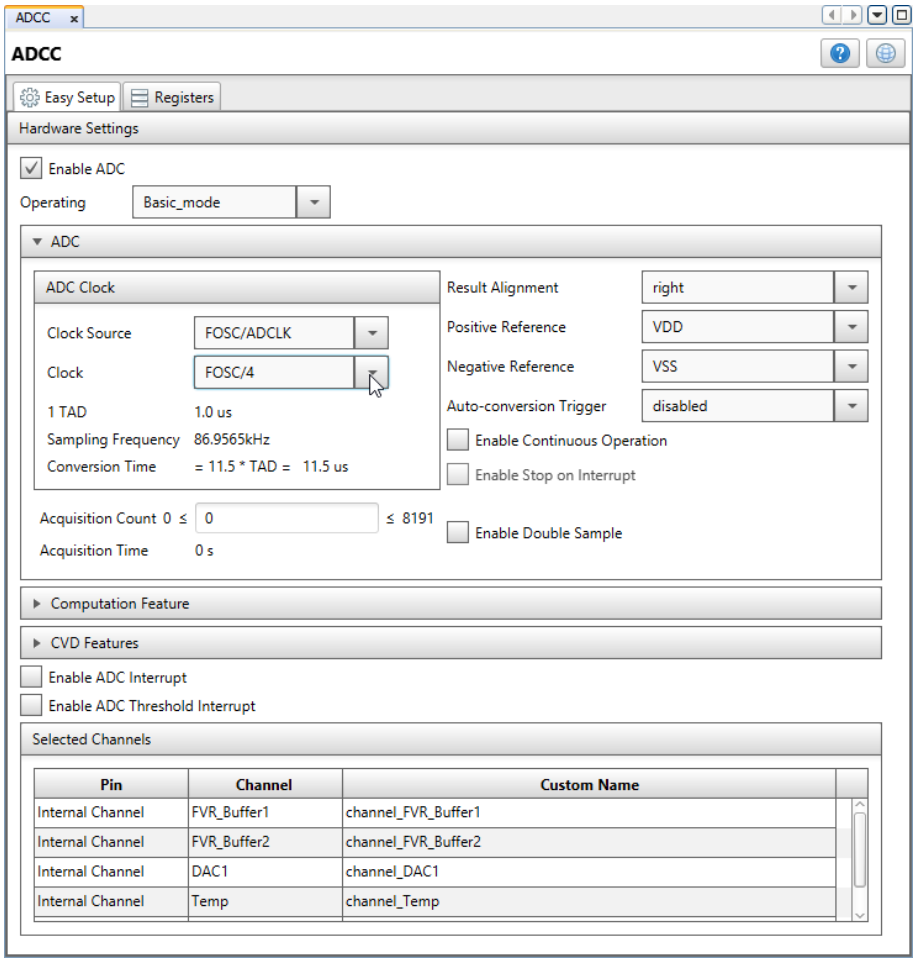



**Figure 14-7:** ADC configuration.

After that configuration, you move to the Pin Manager: Grid View, see Figure 14-8, and decide how many pins you wish to use to configure as analogue pin for voltage measurements. Analogue pins are named ANx in Figure 14-8. The two pins selected would be called ANA4 and ANA5. The second A refers to the port and the number to the position in that port. While there is only one ADC peripheral in this PIC/MCU, you can measure voltages for multiple points. It just has to be done sequentially. Since the ADC is fast, this is not a serious drawback as long as the time variation in the circuit voltages that you are investigating is much slower that the ADC conversion rate. Note, because we used $V_{DD}$ and $V_{SS}$ as our references $V_{REF+}$ and $V_{REF-}$, and because the PIC/MCU is aware on these voltages internally, there is no need to do anything with VREF+ and VREF– in

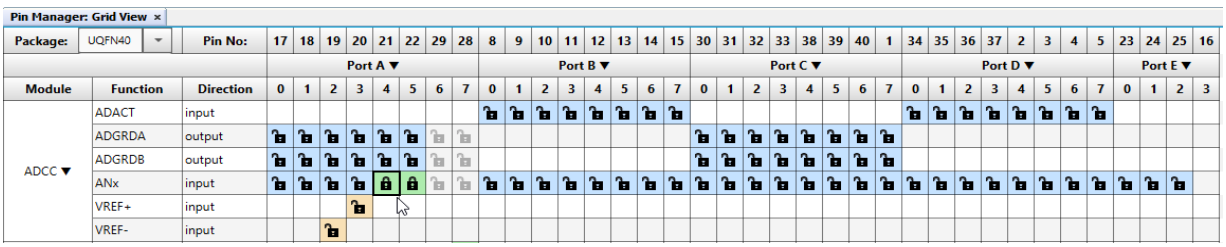



**Figure 14-8:** ADC pin selection.

Figure 14-8. Nor is there any necessity to wire these pins to the breadboard power and ground rails. That is all handled internally.

In the Pin Module tab shown in Figure 14-9 you can create more informative names. Note that the pins are automatically checked as Analog.

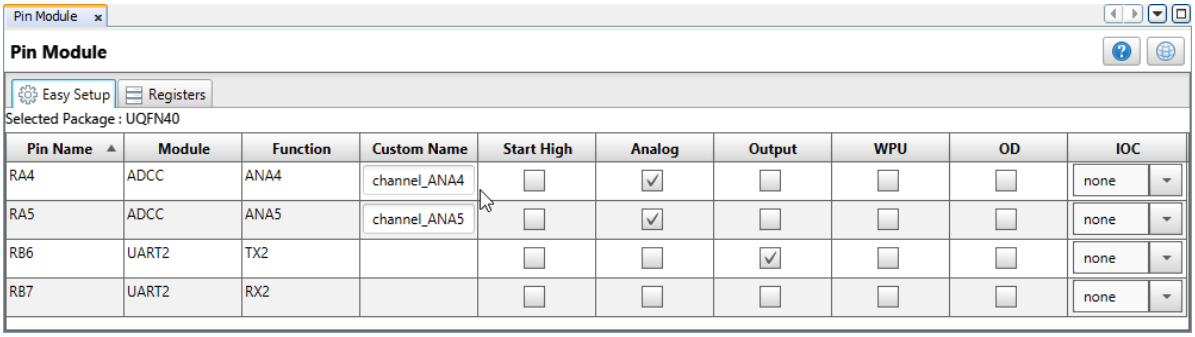



**Figure 14-9:** Custom names for analogue pins.

When MCC generated the code for the configuration, there is only one function that we need to be concerned about. This function is `ADCC_GetSingleConversion(channel_ANAn)` which returns an integer value between 0 and 4095. This function monitors the ADC measurement from start to finish. That means about it will run for approximately one Conversion Time shown in Figure 14-8. The example code shown below, takes consecutive measurements on two pins, ANA4 and ANA5, using this function. The results are converted to millivolts using Equation 14-1 and function `convertADCtoMillivolts()`. This function uses two variables, `vrefplus` and `vrefminus`, placed in `#define` statements before the `main()` function. The voltage of the power rail on your breadboard comes from the USB connecter which is only nominally +5 V. You should use your DMM to measure the actual voltage and use that value before using this program.

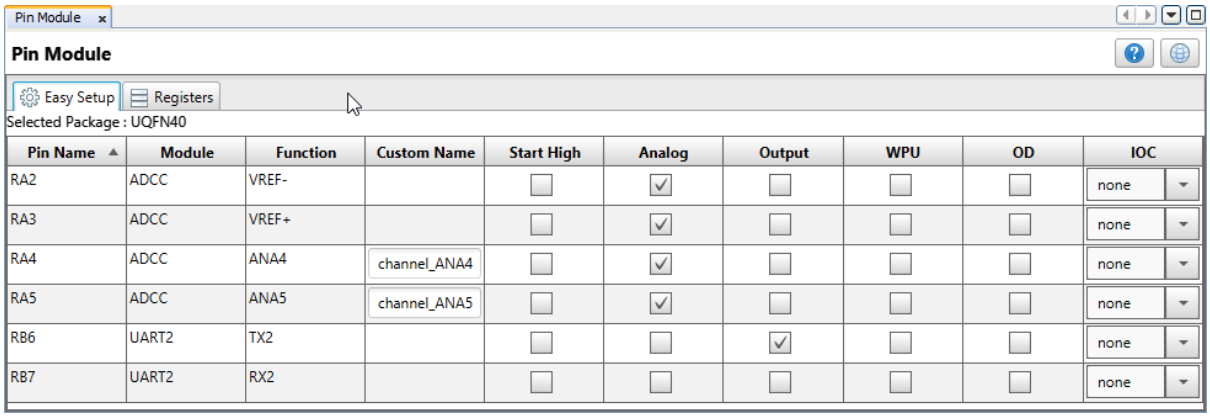
```
// 2ChannelADC.c
// system at 32 MHz
// UART2 at 9600 bps
// ADC Basic Mode, VDD & VSS, Continuous Operation
// RA4 and RA5 as ADC inputs ANA4 and ANA5
// delays

#include "mcc_generated_files/mcc.h"
#include "putty.h"
#include <stdio.h>
#include <stdbool.h>
#include <math.h>

#define vrefplus 5000
#define vrefminus 0

unsigned int convertADCtoMillivolts(unsigned int adcValue);

void main(void)
{
    // Initialize the device
    SYSTEM_Initialize();
```

```
    unsigned int ADCvoltage1, ADCvoltage2;
    adc_result_t convertedValue1, convertedValue2;

    clearPuTTY();

    printf("ADC testing \n\n\r");

    while (1)
    {
      convertedValue1 = ADCC_GetSingleConversion(channel_ANA4);
      convertedValue2 = ADCC_GetSingleConversion(channel_ANA5);

      ADCvoltage1 = convertADCtoMillivolts(convertedValue1);
      ADCvoltage2 = convertADCtoMillivolts(convertedValue2);
      printf("ANA4: %u = %u mV \n\r", convertedValue1, ADCvoltage1);
      printf("ANA5: %u = %u mV \n\n\r", convertedValue2, ADCvoltage2);
      DELAY_milliseconds(250);
    }
}

unsigned int convertADCtoMillivolts(unsigned int adcValue)
{
    return ((unsigned int)((adcValue + 0.5)*
                (float)(vrefplus - vrefminus)/4096.0 + vrefminus));
}
```

## Using External References

Using the internal or board voltages VDD and VSS in a simple choice and gets one measuring voltages fast. You can make other choices which require a little more work. In Figure 14-8, on the right side of the Easy Setup tab, you can set either or both of the *Positive Reference*, $V_{REF+}$, and the *Negative reference,* $V_{REF}$, to *external*. In Pin Manager: Grid View, Figure 14-8, you should then see that whichever of VREF+ and VREF– that are external are now locked to green. The corresponding analogue pins ANA3 (RA3 on the Xpress board) and ANA2 (RA2) become unavailable. When you chose either of VREF+ and VREF– as external references, you must connect the corresponding pin ANA3/RA3/ VREF+ and ANA2/RA3/ VREF– to an actual external voltage. To test it out, you could connect ANA3/RA3/ VREF+ to VDD and ANA2/RA3/ VREF– to VSS. This, in effect, reconstructs the VDD and VSS settings in *Positive Reference* and *Negative reference* in the MCC ADC Easy Setup. That is okay and a good way to check that everything is working correctly.

When external references are chosen, the Pin Module tab changes from Figure 14-9 to Figure 14-10. To get voltages different from VDD and VSS, a *voltage divider* board as shown in Figure 14-11 is used. The voltage divider is just a number of identical resistors in series between power and ground. With two resistors in a voltage divider board, the voltage at the point between the two resistors is just $\frac{1}{2}$VDD. A 3-resistor voltage divider could provide two points, one at $\frac{1}{3}$VDD and one at $\frac{2}{3}$VDD. As you may surmise, a board with *n* resistors gives you voltages from $\frac{1}{n}$VDD to $\frac{n-1}{n}$VDD that you could potentially use for VREF+ and VREF–.




| Pin Name | Module | Function | Custom Name | Start High | Analog | Output | WPU | OD | IOC |
|---|---|---|---|---|---|---|---|---|---|
| RA2 | ADCC | VREF- | | ☐ | ☑ | ☐ | ☐ | ☐ | none |
| RA3 | ADCC | VREF+ | | ☐ | ☑ | ☐ | ☐ | ☐ | none |
| RA4 | ADCC | ANA4 | channel_ANA4 | ☐ | ☑ | ☐ | ☐ | ☐ | none |
| RA5 | ADCC | ANA5 | channel_ANA5 | ☐ | ☑ | ☐ | ☐ | ☐ | none |
| RB6 | UART2 | TX2 | | ☐ | ☐ | ☑ | ☐ | ☐ | none |
| RB7 | UART2 | RX2 | | ☐ | ☐ | ☐ | ☐ | ☐ | none |

**Figure 14-10:** Reference pins in Pin Module.

The advantage of using external references can be seen with an example. Imagine you wish to measure a voltage that you know lies between 1.0 and 2.0 volts. Eqn [14-1] gives a much more precise result with VREF+ = 2.0 V and VREF– = 1.0 V than if VREF+ = VDD and VREF– = VSS by a factor of five.

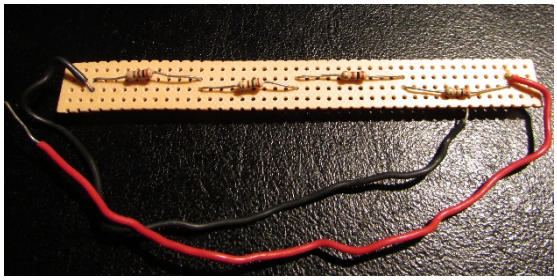


**Figure 14-11:** Voltage Divider.

## Precision Reference

The precision of an ADC measurement is overstated by the uncertainty portion of Eqn [14-1]. It is also determined by how well one knows the difference VREF+ – VREF–. Your board voltage has to be measured by a DMM which has its own measurement uncertainty. Also, that reading can vary if other devices are plugged into the USB ports on your PC and by how much power they need at any particular instant.

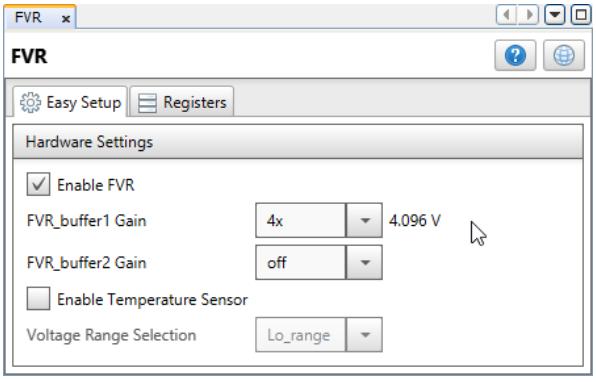



**Figure 14-12:** Configuring FVR.

The PIC/MCU has a *Fixed Voltage Reference* (FVR) module that can be used to gain extra precision. In Easy Setup for the ADC peripheral, Figure 14-8, set the *Positive Reference*, $V_{REF+}$, to FVR. The FVR reference is internal, no wiring is needed. Note that $V_{REF-}$ cannot use the FVR module. The FVR module must be configured in MCC. The Easy Setup tab for FVR is shown in Figure 14-12. ADC uses the *FVR_buffer1 Gain*.

The possible settings are precisely 1.024 V, 2.048 V, or 4.096 V.

## ADC and Transducers

Transducers are electronic devices whose physical properties, usually voltage output or resistance, changes with variations in its surroundings such as temperature, pressure, ambient light, and the like. Often the voltage output is quite small, on the range of millivolts or less, and thus needs to be amplified for most uses.

A photoresistor is an electronic component whose resistance decreases with increasing incident light intensity. It can also be referred to as a light-dependent resistor (LDR), photoconductor, or photocell.

A photoresistor is made of a high-resistance semiconductor. When no light is shining on it, its resistance is very large, as much as 10 MΩ. If light of high enough frequency shines on the device, photons absorbed by the semiconductor give bound electrons enough energy to jump into the conduction band. The resulting free electron (and its hole partner) conduct electricity, thereby lowering resistance. Under bright light, the resistance can drop to about 100 Ω.

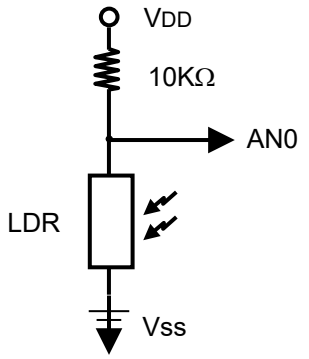



**Figure 14-13:** LDR in a voltage divider circuit.

ADC measures voltages, not resistance, so we would need to put the photoresistor in a voltage divider circuit, Figure 13-11. Under no light, LDR is millions of ohms, so the potential at ANO would be almost $V_{DD}$. Under the brightest conditions, the LDR has very small resistance, so the potential at ANO would be almost $V_{SS}$.

In the lab, you will be working with an infrared (IR) reflective sensor (RLS). The RLS is a package containing an LED that emits IR light and an IR-sensitive phototransistor to detect and amplify any reflected light that falls on the phototransistor. In operation, the phototransistor is much like a photoresistor but the physics is much different. A phototransistor can be thought of as a photodiode and an amplifier. A photodiode is the inverse of an LED. An LED emits light when a current passes through it; a photodiode generates a current when light falls on it. This current is small, so it needs to be amplified. Figure 13-12 shows how the RLS is connected to an ADC pin on the MCU. The right-hand side is much

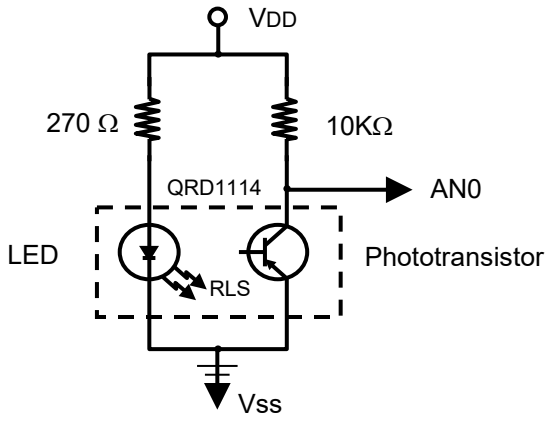



**Figure 14-14:** Connecting a RLS to the Breadboard.

like the LDR in the voltage divider circuit. With no light, there is no current to amplify and the phototransistor is effectively an extremely large resistance. Hence, the voltage at the ADC pin will be large. Conversely, bright light produces the most current, which is then amplified. The resistance of the phototransistor is effectively very small, so the voltage at the ADC pin will be small.

A plot of the measured voltage versus the physical property, such as light intensity, is often a nonlinear curve. Manufacturers may provide a formula, P = f(V), so you can in fact measure the physical parameter P. More often there is only a general curve, and it is necessary to measure one or two known values of the parameter to calibrate the curve. For example, if you have a thermocouple or a thermistor for measuring temperature, then ice water at 0° and boiling water at 100° make excellent references.

Q. What happens if we swap the location of the phototransistor and LED with the 10 KΩ and 270 Ω resistors? Will the RLS still work? What will the ADC values be over white paper and over black tape?

# Chapter 15. Analog Output (DAC)

Digital-to-Analogue Conversion (DAC) allows one to output a voltage with more gradations than the low (VSS or ground) and high (VDD or +5 Volts) of Digital Output. While some chips offer 10 or 12 bits of resolution, the DAC peripheral on the PIC MCU we use only has five bits or 32 values. The output is on either, or both, of the DAC1OUT/RA2 and DAC2OUT/RB7 pins. The voltage output is given by

$$DAC_{output} = \left\{(V_{REF+} - V_{REF-}) \times \frac{setting}{2^5}\right\} + V_{REF-} . \qquad \text{Eqn [15-1]}$$

where *setting* is an integer from 0 to 31. DAC can be considered the inverse function of ADC.

The MLAB Code Configurator (MCC) lets you configure the DAC module from the Easy Setup tab as shown in Figure 15-1. Check the *Enable DAC* and *Enable output on DACOUT1* pin boxes. The simplest values for the *Positive Reference* and *Negative Reference* are VDD and VSS. That means VREF+ and VREF– in Eqn [15-1] would be VDD and VSS. While you can check either or both of the *Enable output on DACOUT1* pin and *Enable output on DACOUT2* pin boxes. DO NOT chose DACOUT2!

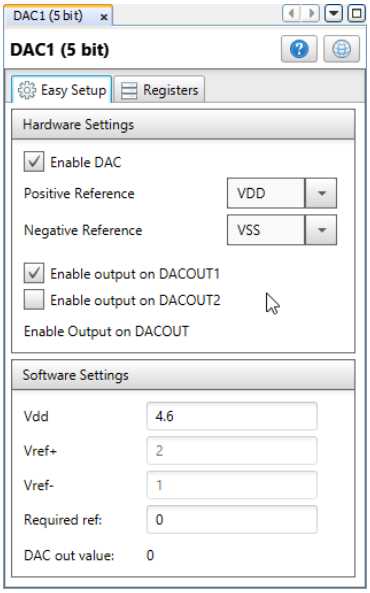



**Figure 15-1:** DAC configuration.

**Warning**: DACOUT2 is multiplexed with RB7. Unfortunately, RB7 is already used by the Xpress Board and cannot function as DAC1OUT2. We are restricted to only using DAC1OUT1/RA2. That will cause other problems, as we will see later.

There are other possible choices for the *Positive Reference* and *Negative Reference*. You could choose *external* for either. When external is chosen, the PIC MCU needs to know the reference voltage. This is accomplished connecting an external voltage to the VREF+/RA2 and/or VREF–/RA3 pins as need.

To get voltages different from VDD and VSS, a *voltage divider* board as shown in Figure 15-2 is used. The voltage divider is just a number of identical resistors in series between power and ground. With two resistors in a voltage divider board, the voltage at the point between the two resistors is just $\frac{1}{2}$VDD. A 3-resistor voltage divider could provide two points, one at $\frac{1}{3}$VDD and one at $\frac{2}{3}$VDD. As you may surmise, a board with *n* resistors

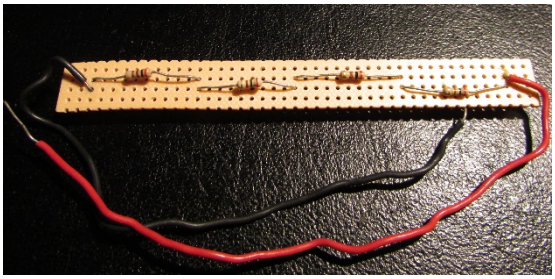


**Figure 15-2:** Voltage Divider.

gives you voltages from $\frac{1}{n}$VDD to $\frac{n-1}{n}$VDD that you could potentially use for VREF+ and VREF–.

The advantage of using external references can be seen with an example. Imagine you wish to output voltages between 1.0 and 2.0 volts. Eqn [15-1] gives more possibilities in that range when VREF+ = 2.0 V and VREF– = 1.0 V than if VREF+ = VDD and VREF– = VSS.

There is a problem with using VREF– as external. If you check Pin Manager: Grid View, Figure 15-3, you will see that VREF– is multiplexed with DAC1OUT1/RA2. This would be okay if we could use DAC2OUT on RB7, but as mentioned above that is not possible with the Xpress Board. So, we cannot use this setting.

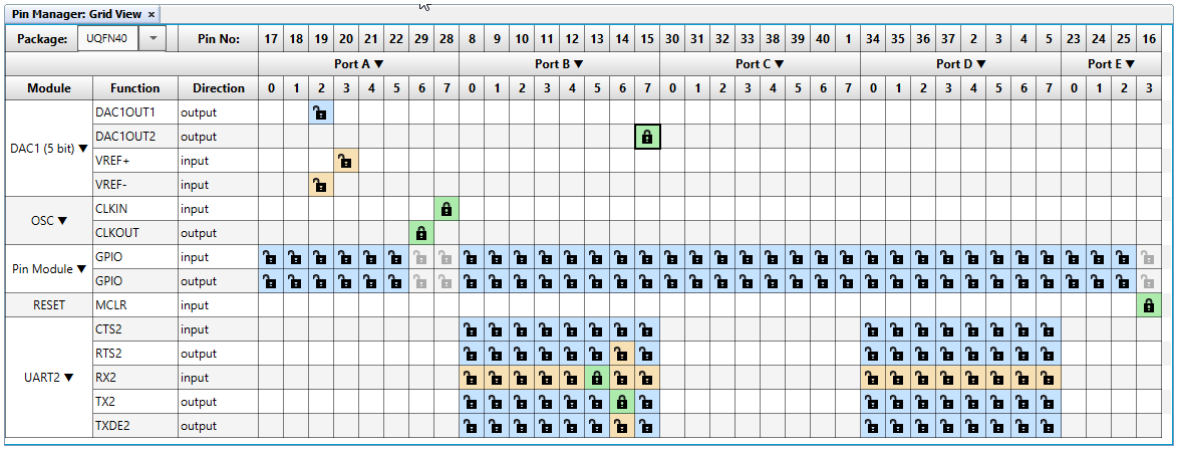



**Figure 15-3:** DAC pin configuration.

There is one last setting to discuss. You may set *Positive Reference* to *FVR_buff2*. The FVR, Fixed Voltage Reference, is an internal setting so no pins have to be wired. In the FVR Easy Setup tab, Figure 15-4, *FVR_buffer2 Gain* can be set to 4.096 V, 2.048 V, or 1.024 V. The advantage of this setting is the precision of the FVR voltage compared to VDD.

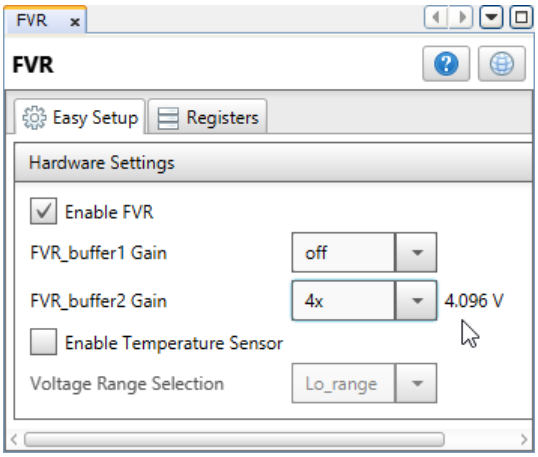



**Figure 15-4:** FVR configuration.

When the MCC files are generated, there are several useful functions created for use with the DAC module. The first is `DAC1_SetOutput(setting)`, where `setting` is a number from 0 to 31 and which changes the voltage output on the active DAC1OUTx pin. The second is `DAC1_GetOutput()` which returns the current value of `setting`, a number between 0 and 31. Certain Special Function Registers (SFRs) change be used to either read of change the DAC and FVR settings, see Table 15-1 and Table 15-2. Example code is shown below. The start of the program just reads the current settings of the DAC and FVR peripherals. Inside the while loop, all possible values of output voltage are generated, one every 5 seconds. A function to convert from DAC setting to actual voltage in millivolts is also used.

**Table 15-1:** DAC Functions and SFRs

| DAC1 Function and SFRs | Description |
|---|---|
| void DAC1_SetOutput(uint8_t setting) | Set output pin voltage. Setting is a number from 0 to 31. Voltage determined by Eqn 15-1 |
| uint8_t DAC1_GetOutput(void) | Reads the setting, a number from 0 to 31. |
| DAC1CON0bits.OE1 | Read/Write. DAC1OUT1/RA2 active. 0 = No, 1 = Yes |
| DAC1CON0bits.OE2 | Read/Write. DAC1OUT2/RB7 active. 0 = No, 1 = Yes |
| DAC1CON0bits.PSS | Read/Write. DAC positive reference.<br>Value Selection<br>0 VDD<br>1 External<br>2 FVR buffer 2 |
| DAC1CON0bits.NSS | Read/Write. DAC negative reference.<br>Value Selection<br>0 VSS<br>1 External |

**Table 15-2:** FVR SFRs

| FVR SFRs | Description |
|---|---|
| FVRCONbits.RDY | Read only. Is FVR in use? 0 = No, 1 = Yes |
| FVRCONbits.ADFVR | Read/Write. Buffer 1 setting<br>Value Selection<br>0 FVR off<br>1 1.024 V<br>2 2.048 V<br>3 4.096 V |
| FVRCONbits.CDAFVR | Read/Write. Buffer 1 setting<br>Value Selection<br>0 FVR off<br>1 1.024 V<br>2 2.048 V<br>3 4.096 V |

```
// DAC.c
// system at 32 MHz
// UART2 at 9600 bps
// DAC using VDD & VSS
// DAC1OUT/RA2
// delays

#include "mcc_generated_files/mcc.h"
#include "putty.h"
#include <stdio.h>

#define vrefplus  4816
#define vrefminus 0

unsigned int DACsettingtoMillivolts(unsigned int DACsetting);

int main(void) {
```

```
 // Initialize the device
 SYSTEM_Initialize();
 int i;

 clearPuTTY();

 printf("DAC out on RA2 \n\n\r");

 printf("DAC Settings\n\n\r");
 printf("Is DAC1OUT1/RA2 used = %u (0 = N, 1 = Y) \n\r", DAC1CON0bits.OE1);
 printf("Is DAC1OUT2/RB7 used = %u (0 = N, 1 = Y) \n\r", DAC1CON0bits.OE2);
 printf("Positive Source = %u (0 = VDD, 1 = external, 2 = FVR) \n\r",
                           DAC1CON0bits.PSS);
 printf("Negative Source = %u (0 = VSS, 1 = external) \n\n\r",
                           DAC1CON0bits.NSS);

 printf("FVR Settings\n\n\r");
 printf("Is FVR used = %u (0 = N, 1 = Y) \n\r", FVRCONbits.RDY);
 printf("Buffer_2 = %u (0 = Off, 1 = 1.024 V, 2 = 2.048 V,
                     3 = 4.096 V) \n\n\r", FVRCONbits.CDAFVR);

 printf("vref+ = %u mV, vref- = %u \n\n\r", vrefplus, vrefminus);

 while(1){
   i = 0;
   for(i=0; i < 32; i++)
   {
    DAC1_SetOutput(i);
    printf("i = %2u, V_out = %4u mV \n\r", i, DACsettingtoMillivolts(i) );
    DELAY_milliseconds(5000);
   }
 } // end while
}

unsigned int DACsettingtoMillivolts(unsigned int DACsetting)
{
    return ((unsigned int)(DACsetting*(float)(vrefplus - vrefminus)/32.0 +
vrefminus));
}
```

# Appendix A. Software

The software we will use to write, compile, and debug C language code and to program the PIC MCU are free courtesy of Microchip™, the manufacturers of the PIC18F46K42 chip that we are using in this course. The software:

*MPLABX IDE (Integrated Debugger Environment), and*
*MPLABX XC8 Compiler*
*MPLAB X Master Code Configurator*
*MPLAB X Simple Serial Port Terminal*

as well as a wealth of support documentation in PDF format :

*MPLAB™ XC8 C Compiler User's Guide for PIC MCU,*
*MPLAB™ XC8 C Compiler Libraries,*
*MPLAB Code Configurator v3.xx User's Guide,*
*MPLAB™ IDE User's Guide,*
*MPLAB™ IDE Quick Start Guide,*
*PIC18F46K42 Data Sheet*

are available from www.microchip.com. Download them for use on your home PC if you wish. You will find links on the Physics 1600 Moodle home page.

The MPLABX IDE and XC8 Compiler work together as one seamless program from the view of the user. In the IDE, you build a project. A project consists of a C language code file that you write, and library files supplied by Microchip. A library is a group of functions grouped together for easier reference and ease of linking. You can edit, debug, or step through your code line by line. When your project is ready, you instruct the IDE to compile your program to a .HEX file. The Hex file is your project in a form that the PIC18F46K42 chip can understand. You then copy and paste that Hex file to the XPRESS board that has the PIC18F46K42 MCU. The XPRESS board appears as a new drive on your PC.

## Installing the Programs at Home

The versions of the programs we use, as well as the support documentation, are available for download as link from the Physics 1600 Moodle website. The programs may be installed on your home PC if it is operating Windows OS, Linux OS, or Mac OS. Videos of the installation process on Windows are also on the Moodle site. First install the MPLABX IDE. When it finishes, it will prompt you to choose a compiler. Install the XC8 Compiler (again there is a video on the Moodle site). The MPLAB X Master Code Configurator and MPLAB X Simple Serial Port Terminal are installed from inside the IDE (also on video).

During the install process for the IDE, you will encounter one or more configuration windows. Generally, the default choices work fine. There are several for MPLAB X that will reduce the size of the installation on your PC. In Figure A-1, choose only to install the IDE and XC8 support.

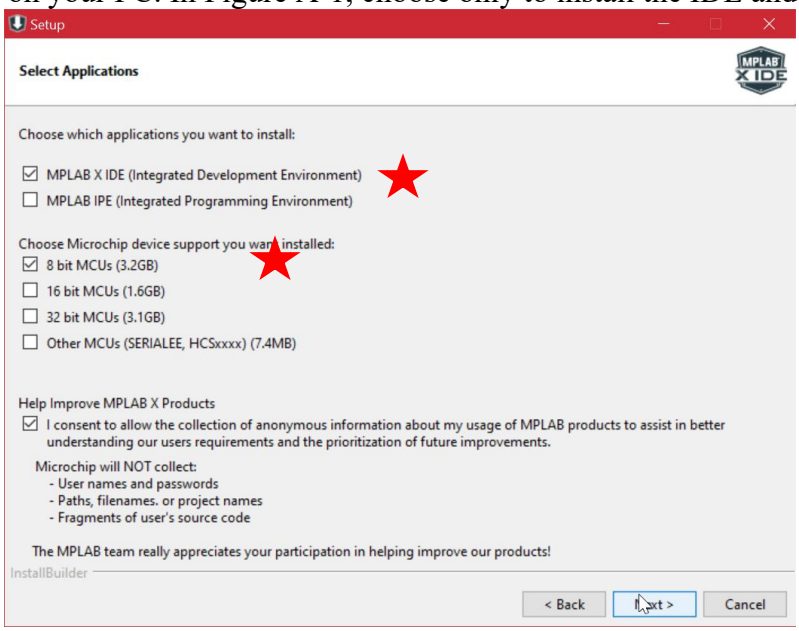



**Figure A-1:** Minimizing the IDE installation.

At the end of the installation of the MPLAB X IDE, a window will pop up asking if you wish to install further programs. Check the XC8 Compiler as shown below in Figure A-2

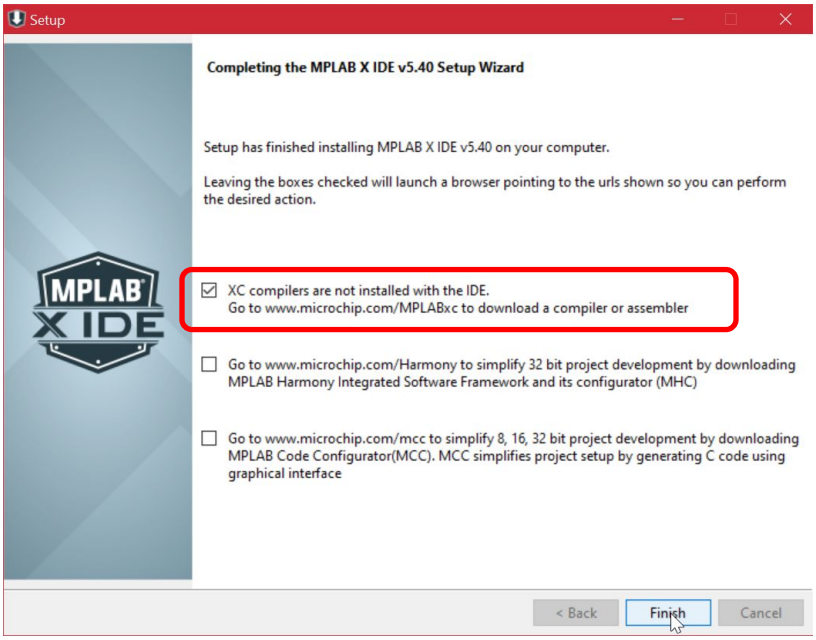



**Figure A-2:** Follow link to XC8 Compiler installation.

If you miss it, do not worry. You can always download the XC8 compiler using the link on the course Moodle site. During the installation of the XC8 compiler, the default choices are fine.

## Plug-ins

Plug-Ins are smaller, helper programs that extend the capabilities of the main program. There are three plug-ins that we will install: the MPLAB X Code Configurator (often abbreviated as MCC), the Simple Serial Port Terminal, and Data Visualizer. Installation is done inside MPLAD X IDE. Open the MPLAB X IDE app if it is not already open and navigate to Tools, then Plugins, as shown in Figure A-3 below.

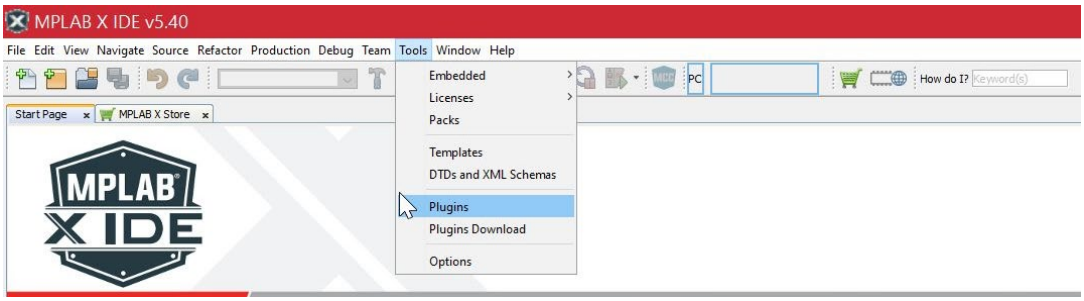



**Figure A-3:** Install Plug-Ins

This will bring up a Plugins window. Search through the list, find and check MPLAB X Code Configurator, Simple Serial Port Terminal, and Data Visualizer as shown in Figure A-4, then install. The MPLAB X IDE will need to shut down and restart.

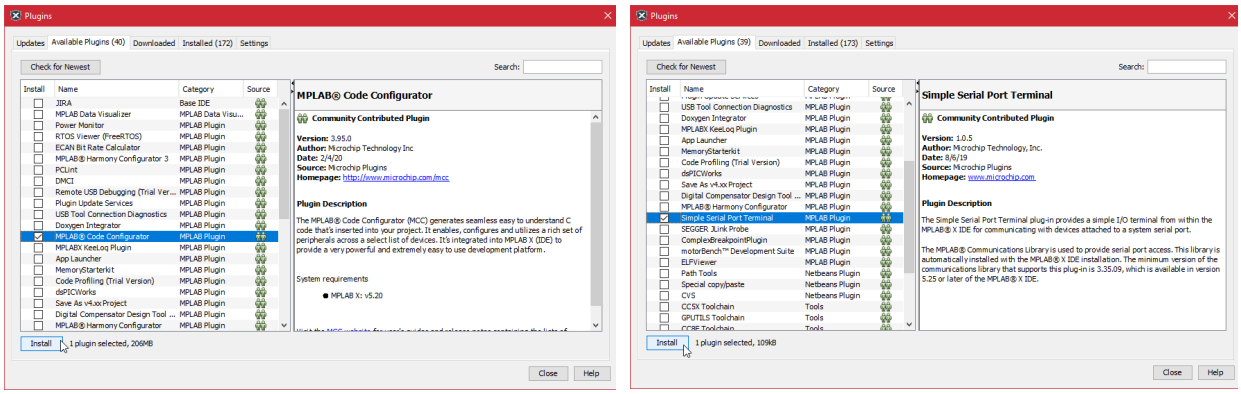



**Figure A-4:** Selecting Plug-Ins to install

When the IDE restarts, the only noticeable difference is that there is now a Data Visualizer and an MCC icon in the icon bar as shown in below Figure A-5 and an icon for Data Visualizer.

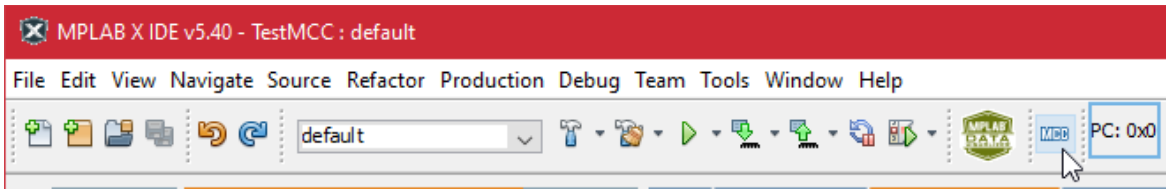



**Figure A-5:** MPLAB X IDE with MCC icon

It is a good idea to check that MCC, the Simple Serial Port Terminal, and Data Visualizer have been installed. Click on Tools then Embedded as shown in Figure A-6 below.

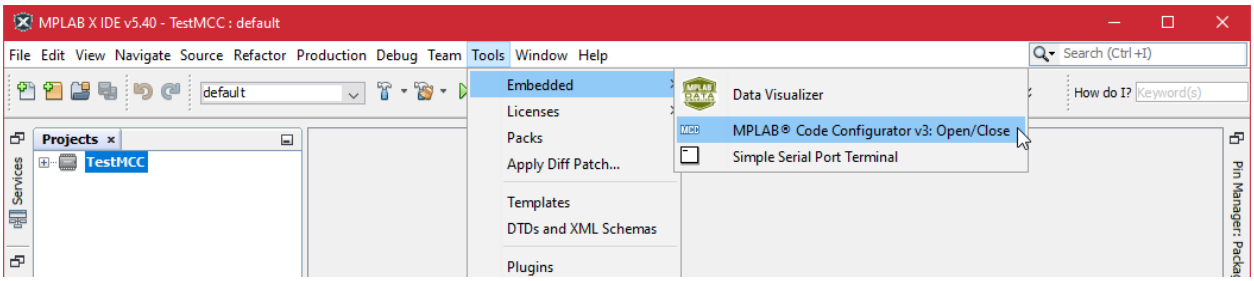



**Figure A-6:** Available Plug-Ins

## Creating a Project

A project is a collection of files, some of which are yours and some of which are standard C library files, that the compiler will turn into a program that will run on the MCU. Typically, you will have several source (.c and .h) files. Only one source file can have a `main()` function or an error will occur when you try to compile your project. If you will be working both at home and in the lab, it is wise to have a USB memory stick and to use it to store your projects. Be sure to back up your files at home since misplacing a memory stick is a common occurrence. Use the default folder called *MPLAB X Projects* to contain your projects. Each project should have its own subdirectory. During the course of the labs, you will be building up a library of useful functions that can be added to any project. Place these functions, and any functions we give you, in a subdirectory of the *MPLAB X Projects* folder called Common.

> **Possible Pitfall**: You can build your project in the Student Directory on the lab PC but files in the Student Directory are deleted every night or sooner. Do not leave your project there. Backup to a USB device.

To start, click on the MPLAB X IDE icon on the desktop of your PC. The IDE opens to a Start Page (basically an advertisement for Microchip). This tab has a box at the bottom that you can uncheck to have it never show again. Close the Start Page tab to go to the work area shown below in Figure A-7. In instructions, I will refer to the choices in the Menu Bar and the shortcuts on the Icon Bar such as the New Projects icon. To create your first project, choose File | New Project from the menu bar or click on the New Project icon or use the keyboard shortcut CTRL+SHIFT+N.

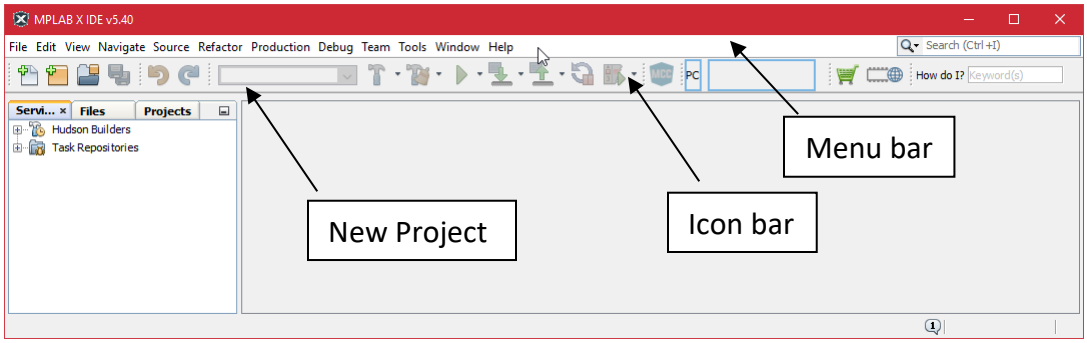



**Figure A-7:** The MPLAB IDE main window (Start Page not showing).

You will create a standalone project as shown in Figure A-8. In the next window, Figure A-9, choose *Advanced 8-bit MCUs (PIC18)*, *PIC18F46K42*, and *No Tool* from the dropdown menus. In the next window, Figure A-10, choose the most recent version of the XC8 compiler. Finally, in Figure A-11, choose the name for the project and the location for it to reside.

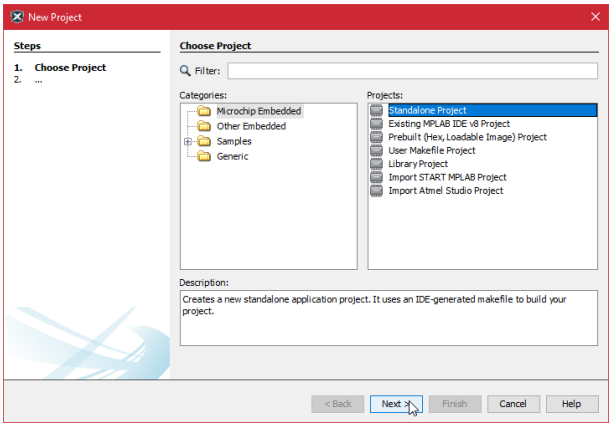



**Figure A-8:** Create Standalone Program

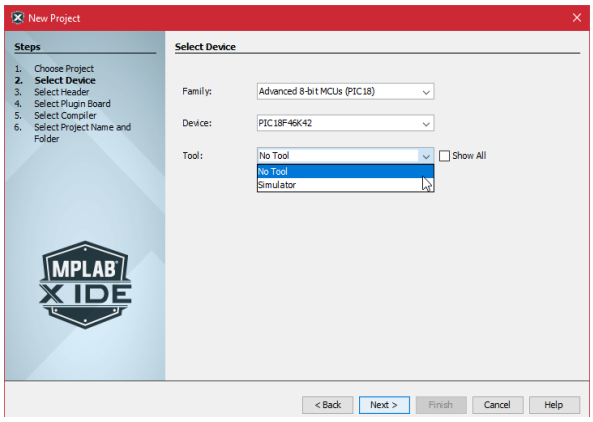



**Figure A-9:** Choose Device and No Tool

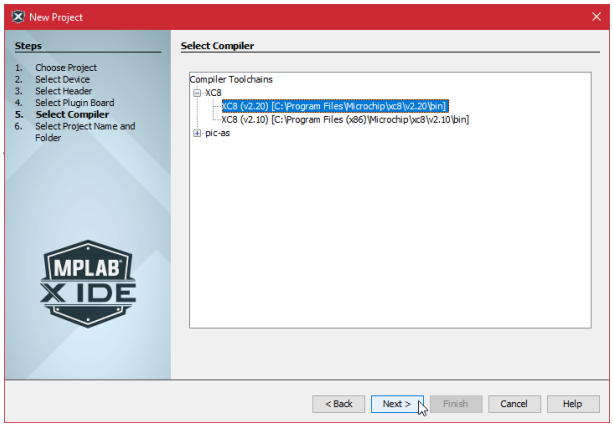



**Figure A-10:** Chose Compiler Version.

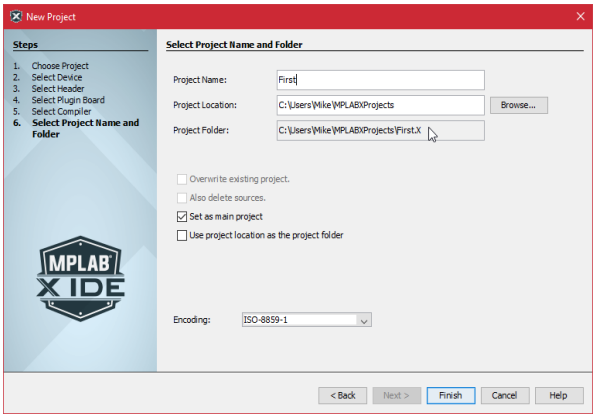



**Figure A-11:** Name Project and Set Location

These actions will have created a shell project shown in Figure A-12 . It still needs a code containing the main() starting function. Further, you need to configure your PIC to work with your device or with the Simulator which is discussed in the next appendix. To do both, use the MPLAB Code Configurator (MCC) plug-in by clicking on the icon in the Icon Bar.

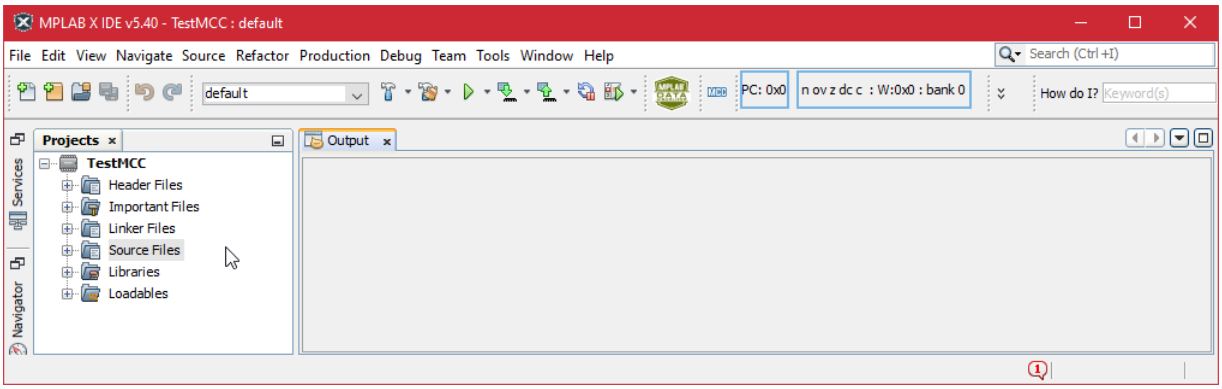



**Figure A-12:** Project shell.

This will create a new tab called *Resource Management [MCC]* on the left-hand side of the IDE, Figure A-13. Under it will be *Project Resources* and *Device Resources*, where you can pick the PIC/MCU peripherals you wish to use. To the right, in the main window is the System Module where you set the operating speed of the PIC/MCU. More on operating speed is discussed in Chapter 9, but for now choose HFINTOSC in the *Oscillator Select* dropdown box, 32 MHz in the *HF Internal Clock* dropdown box, and 1 in the *Clock Divider* dropdown box.

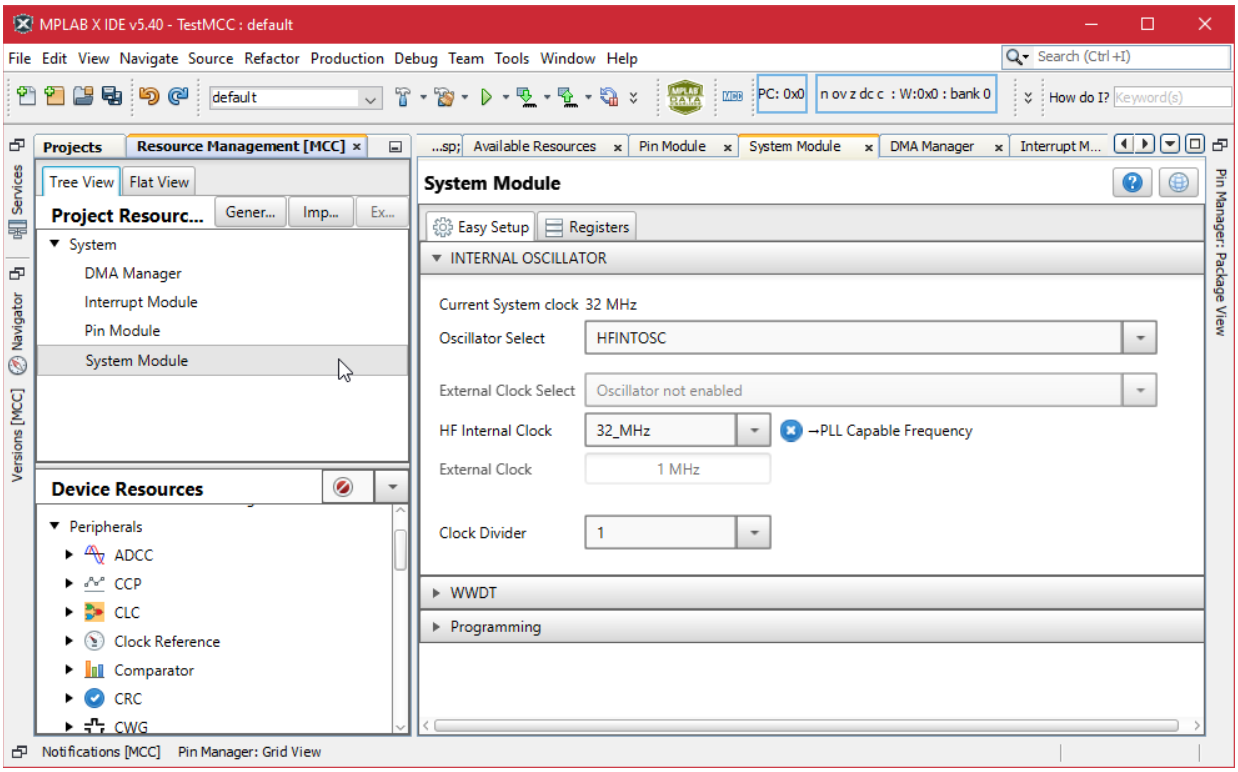



**Figure A-13:** Project Resources and Devices and System Module

Next, we must to select and configure the UART2 peripheral or Module to allow for communication between the PIC/MCU and a terminal program on your PC or with the Simulator discussed in Appendix B. Under *Device Resources* move down the list until you find UART, click on it and then click on the green plus sign next to UART2 as shown in Figure A-14. This will change the look of the IDE. On the left, UART2 is now in *Project Resources* and on the right, we can configure the UART2 peripheral. The defaults here are fine except you must check the box labelled *Redirect STDIO to UART* as shown in Figure A-15.

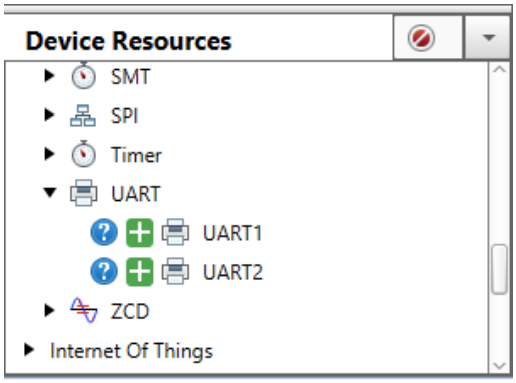



**Figure A-14:** UART2 selection.

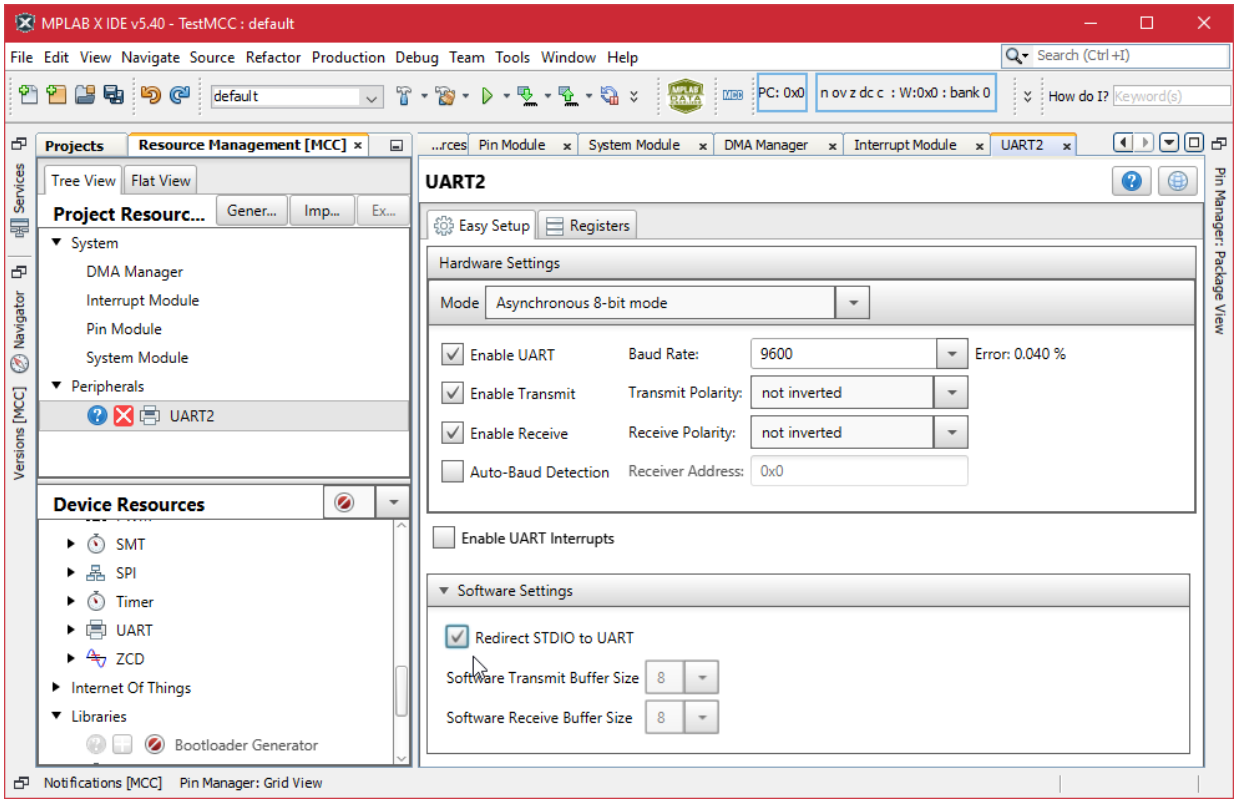



**Figure A-15:** UART2 configuration.

You also need to configure the pins that the UART2 module will use. At the bottom of the IDE, on the left, you will see two minimized windows. Click on *Pin Manager Grid View*. This will open the window shown in Figure A-16. At the bottom of this window click on the blue cell that is at the intersection of row UART2|TX2|output and column Port B|6. It will turn green and show a locked icon.

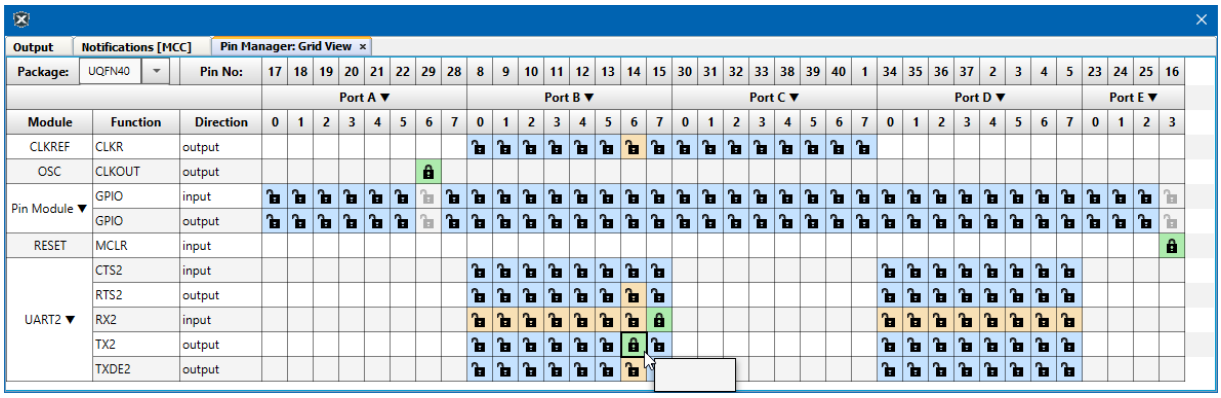



**Figure A-16:** UART2 pin selection.

Next click on the *Generate* tab next to *Project Resources* on the left-hade side of the IDE. In the main window, the *Output* tab in Figure A-16 will show the progress of MCC as it creates the configuration file MyConfig.mc3.

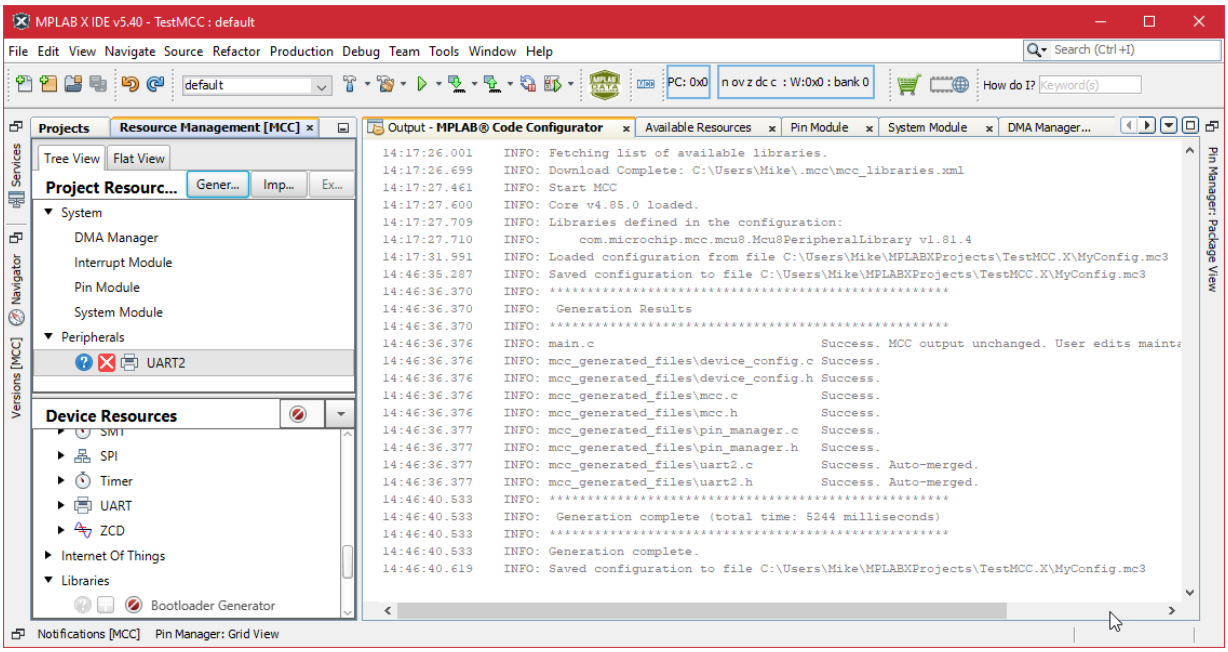



**Figure A-17:** MCC configuration has been generated.

Files generated by MCC have now been added to your project. Click on the *Projects* tab on the left of the IDE and expand the Header Files and Source Files directories by clicking on them. Also click on the main.c source file to show its contents in the main window of the IDE. The expanded view of the project and the contents of main.c are shown in in Figure A-18. I have taken the liberty of deleting all the unnecessary comments from the main.c file. What is left are the key elements: the main() function, a function `SYSTEM_Initialize()` that programs the PIC/MCU to the configuration you previously selected in MCC, and a #include statement to the directory that contains the code for `SYSTEM_Initialize()`.

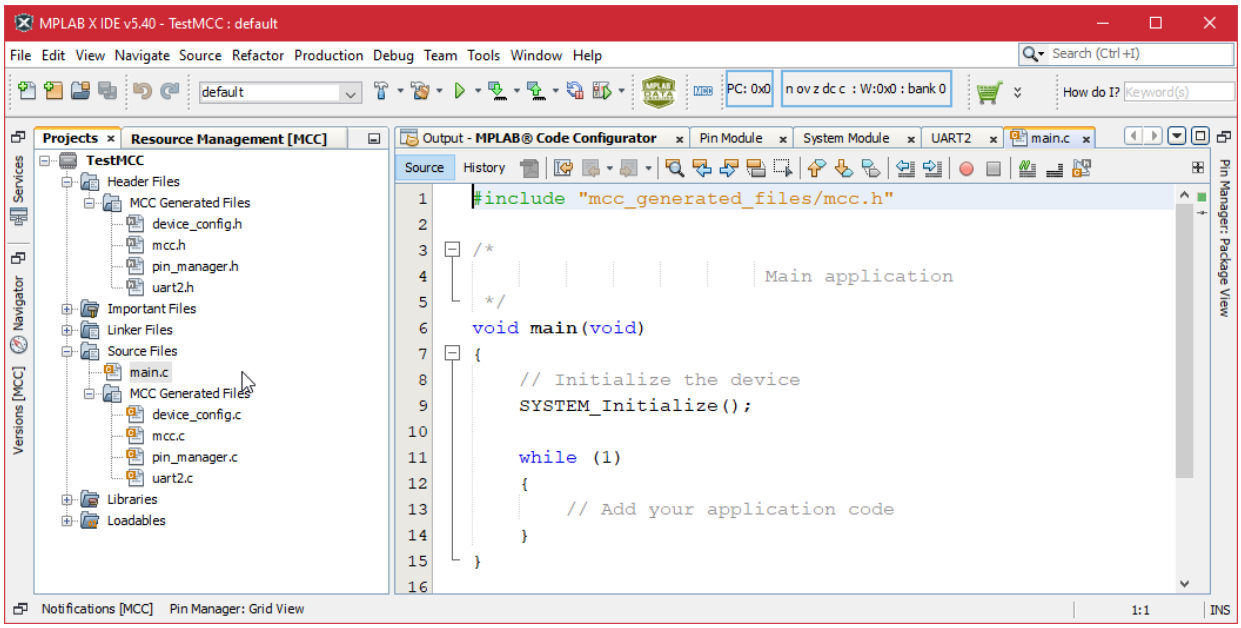



**Figure A-18:** Generated files.

It is worth look at the directory you created to hold this project. As shown in Figure A-19, the is main.c, MyConfig.mcs, and a directory mcc_generated_files. You will likely have to add, or create, other files when you create your projects.

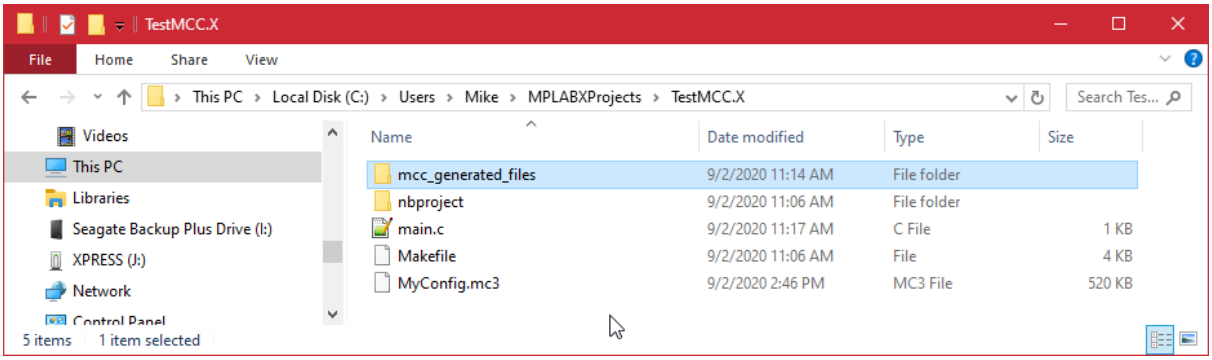



**Figure A-19:** New project files.

To compile this project into a form that can be written to your PIC/MCU, the project must be built by the XC8 Compiler. From the menu bar. choose *Production|Clean and Build Main Project* (shown in Figure A-20). There is an icon on the icon bar and a shortcut Shift+F11 as well.

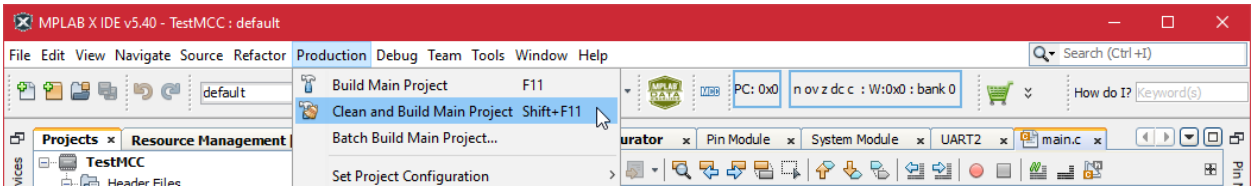



**Figure A-20:** Build and Clean project.

At the project compiles, XC8 will generate a series of warnings (mostly that you have not used all the functions MCC has created for you that you can ignore) as shown in Figure A-21. Some warnings are more serious and you will encounter them later. You may also get error messages. These are serious and must be corrected before the project will compile successfully. Error messages usually indicate on which line of which file the error occurred. The most common errors are incorrect spelling, missing semicolons at the end of a statement, missing terminal brackets, and the like. Some of the more cryptic messages are harder to understand and therefore correct. When all the errors have been corrected, you will be rewarded with a Build Successful message. If not error messages are present, but the build fails, you may need to run *Clean and Build Project* several more times. More information on the error and warning messages is found in *Appendix B* of the *MPLAB XC8 Compiler User's Guide for PIC MCU*. Although more detail is given here, it, too, may be rather cryptic.

**Possible Pitfall**: Sometimes the error in your code is a line or two above the line number specified in the error message.

**Possible Pitfall**: Do not ignore warning messages even if the build succeeds. Subtle logic problems indicated by the warning message can make your program behave in unexpected and undesirable ways.

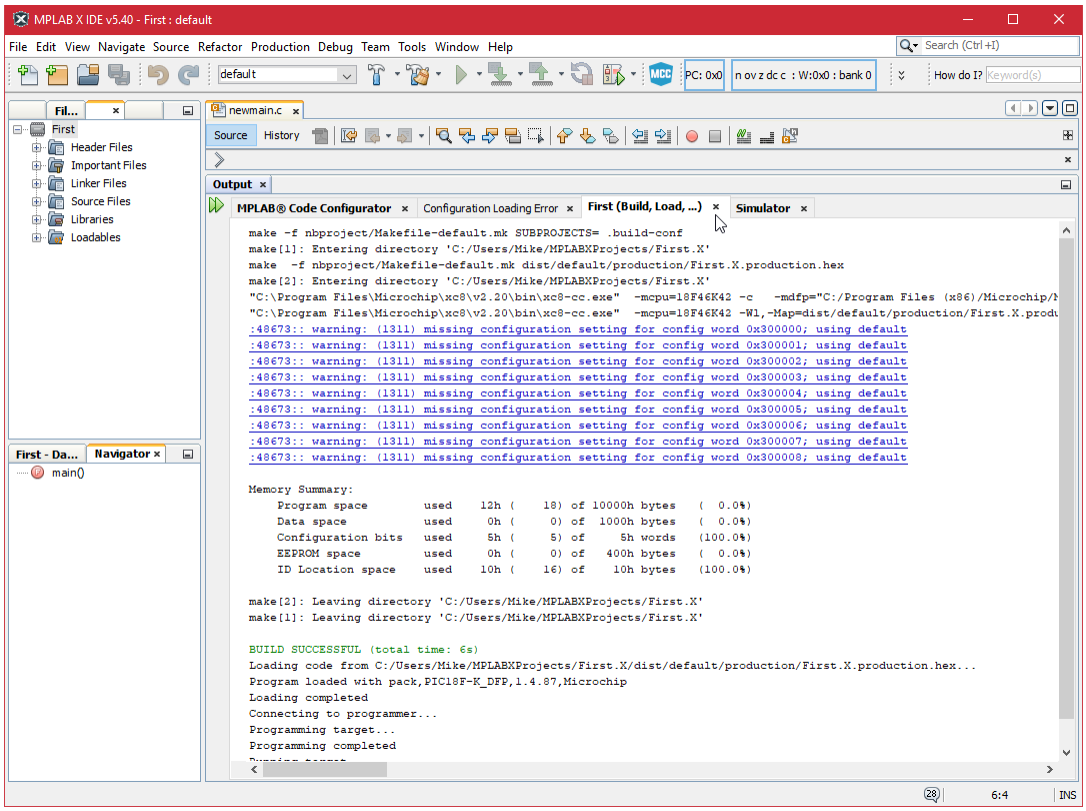



**Figure A-21:** Successful build.

The last step, if you have an Xpress Board plugged into a USB port on your PC, is to export, Figure A-22 the .hex file. A hex file is the only type of file that a PIC/MCU can read. Save the hex file into the memory of the PIC18F46K42 as shown in Figure A-23. Your code is now working but nothing visible will happen because your code only configured the PIC/MCU.

If you do not have an Xpress Board, the Simulator will let you see how the program should work. Refer to the appendix for details. Even in the Simulator, nothing much happens. Writing code that does something is the content of the material in the chapters of this manual.

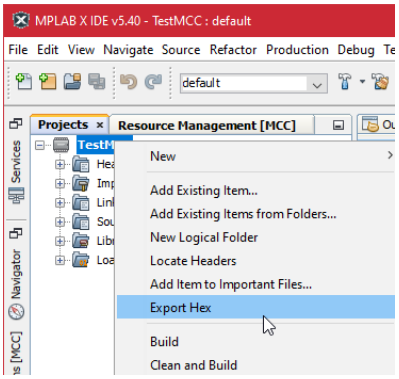



**Figure A-22:** Export Hex

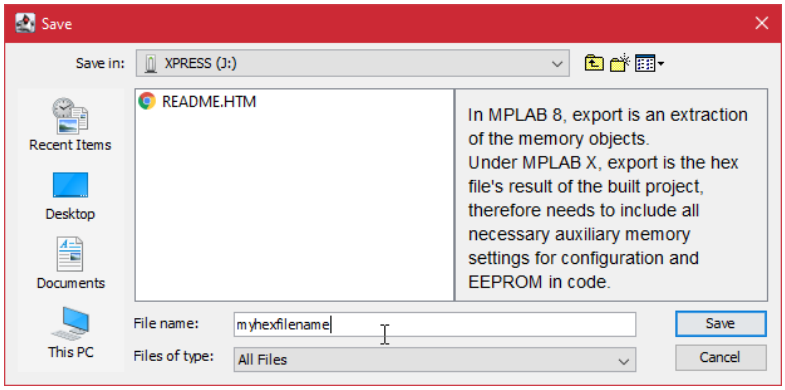



**Figure A-23:** Save hex in Xpress Board

## Timesaver

The job of exporting the hex file to the Xpress board can get quite tedious when you are debugging a program for the 20th time. There is a way to have the last two steps done automatically when you build your project. First open *Project Properties* from the *File* dropdown menu as shown in Figure A-24.

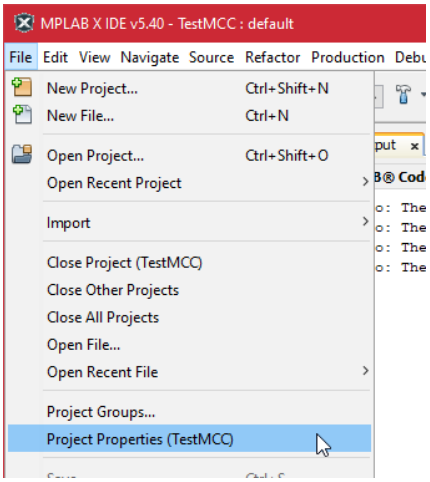



**Figure A-24:** Project Properties

When the *Project Properties* window opens, click *Building* on the left-hand side of the window in *Categories* as shown in Figure A-25. On the right- hand side, check the box labelled *Execute this line after build*. Then in the blank line beneath that box, type cp ${ImagePath} J:\yourfilename.hex. My Xpress Board shows as Drive J: on my PC, so I have J: in the pathname. Use the letter the Xpress Board uses on your PC in that blank line. And change the name of the file to something meaningful. Click on Apply and when a build is successful, the hex file will automatically be sent to the Xpress Board.

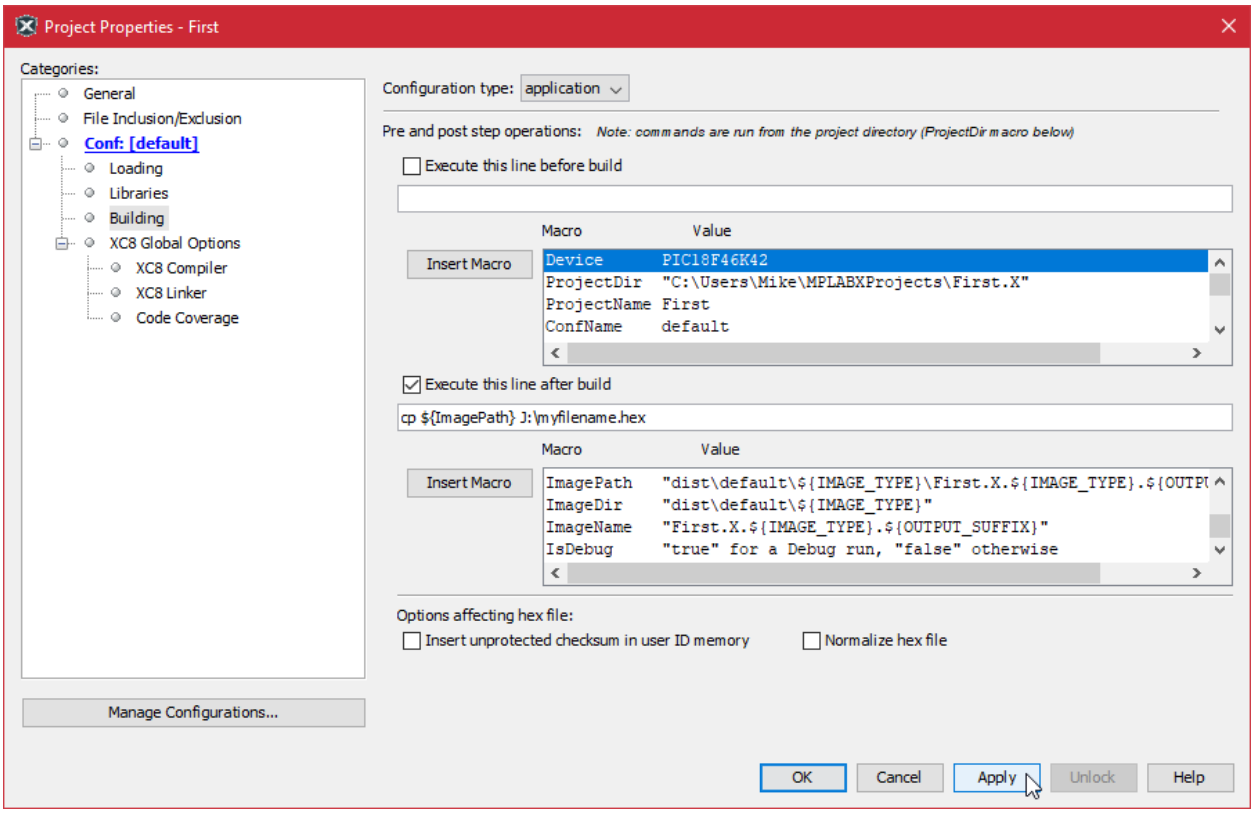



**Figure A-25:** Project Properties

## C90 or C99

The XC8 Compiler are use one of two different versions of C, C90 and C99. C99 is newer and has more features. It is largely backwards compactible with the older and widely tested and trusted C90. Occasionally, strange errors may occur in on version and not the other and you may want to try switching between versions. Again, access *Project Properties* as shown in Figure A-24. When the *Project Properties* window opens, click *XC8 Global Options* on the left-hand side of the window in *Categories* as shown in Figure A-25. On the right-hand side you can choose which *C Standard* to use as shown in Figure A-26.

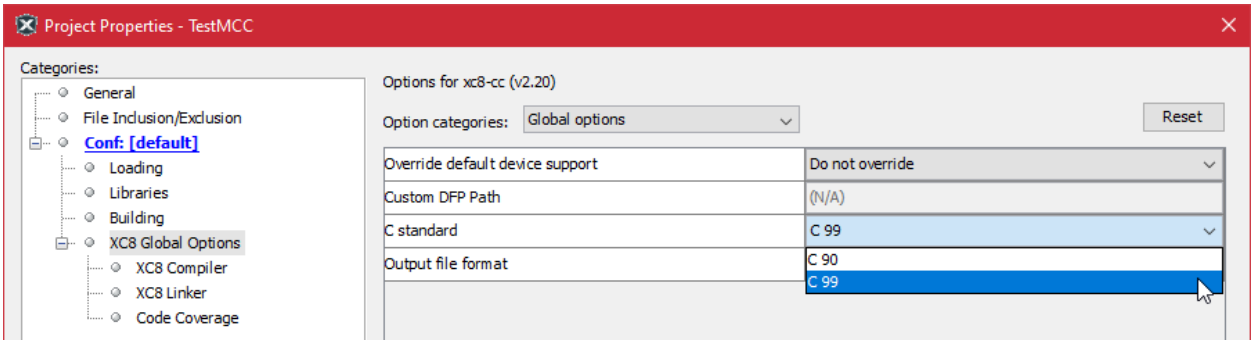



**Figure A-26:** C90 and C99 option.

# Appendix B. The Simulator

The Simulator lets you build, test, and debug programs even if you do not an Xpress Board. Most of the examples in the first few chapters of this manual work fine either in the Simulator or in the PIC/MCU. The misbehaving program may be easier to debug in the Simulator. For example, a program may build successfully but not do what you thought it should because of faulty logic in the way it was constructed. Such faults are much harder to track down, but the Simulator provides a way to step through your code and watch how variables and Special Function Registers (SFRs) change. To invoke the simulator, click on the *Debug Main Project* icon in the Icon bar as in Figure C-1, or use *Debug|Debug Main Project* from the Menu bar as in. There will be a Window pop up, Figure C-2, that asks you to choose the Simulator as the debugger. A new set of icons, Figure C-3, will appear on the icon bar. These icons are shortcuts for stepping through the lines of your program. The first icon in the simulator toolbar, a red square, halts the simulation. The green arrow starts the simulation. The yellow icon pauses the simulation. The purple icon resets the simulation from the beginning.

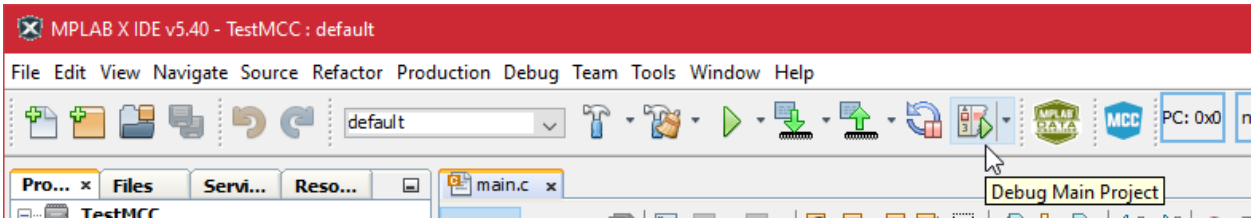



**Figure B-1:** Invoking the Simulator.

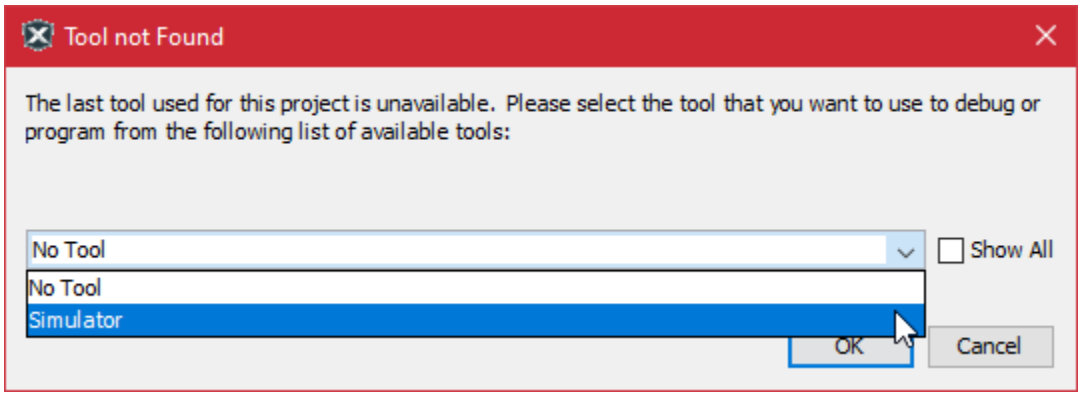



**Figure B-2:** Choosing the Simulator.

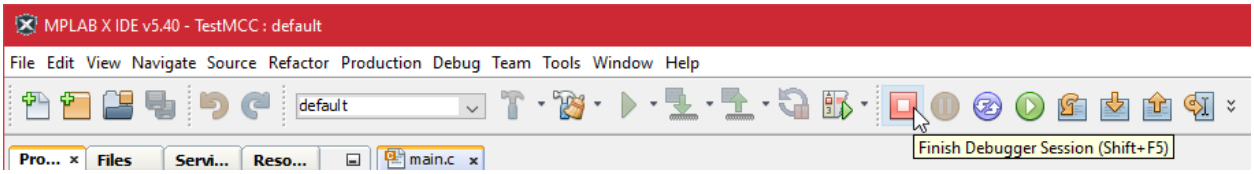



**Figure B-3:** The Debugging Icons

When we create even the simplest projects, we set the system frequency and add the UART2 peripheral. We have to tell the Simulator of these choices. First open *Project Properties* from the *File* dropdown menu as shown in Figure A-24. When the *Project Properties* window opens, click *Simulator* on the left-hand side of the window in *Categories* as shown in Figure C-4. On the right-hand side, fill in the *Instruction Frequency* box with frequency chosen in MCC (probably 32 MHz). Next, in the *Option Categories*, choose *UART2 IO Options* as in Figure C-5. Finally, check the box *Enable UART2 IO* and have the output sent to a window as shown in Figure C-6.

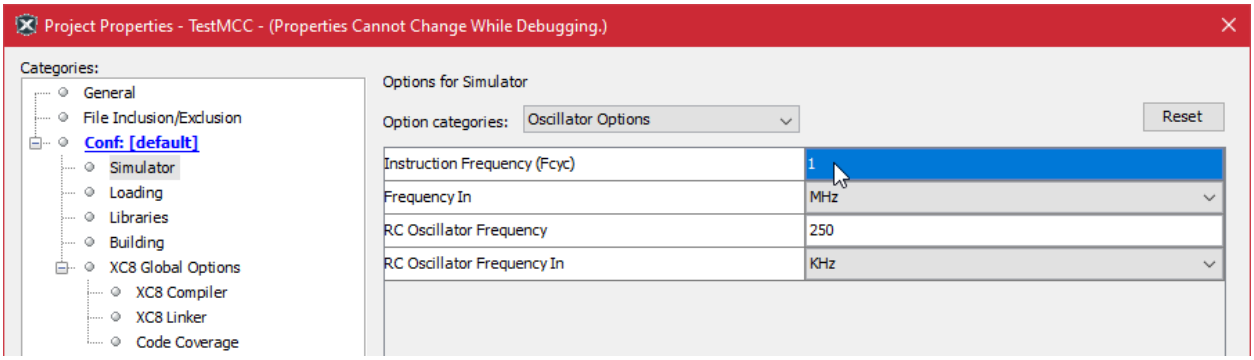



**Figure B-4:** Simulating System frequency

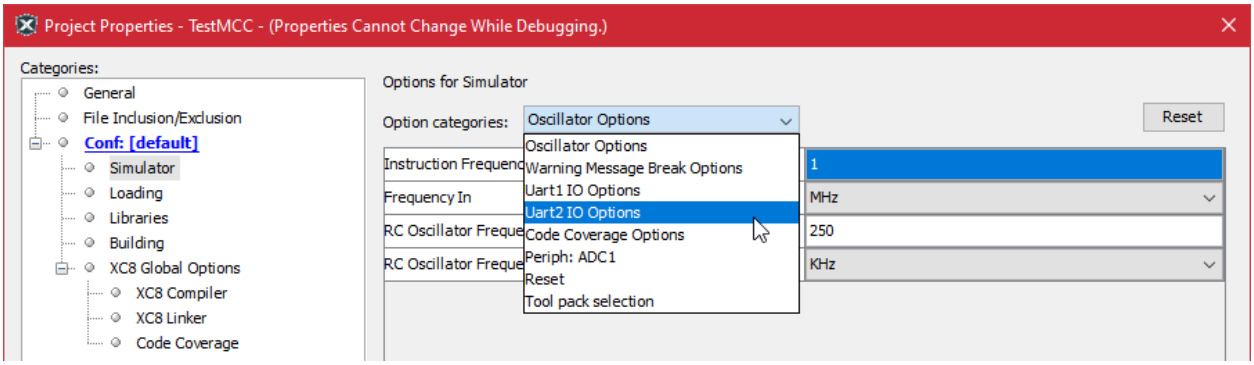



**Figure B-5:** Simulating a terminal window

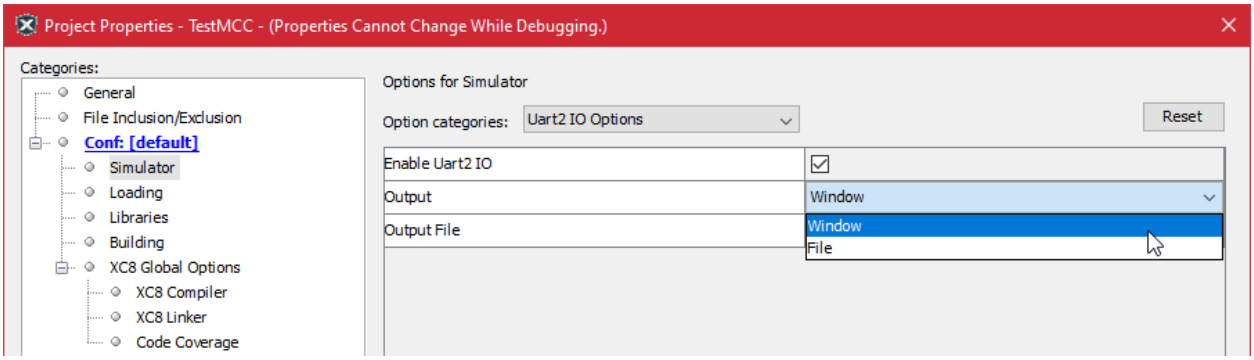



**Figure B-6:** Simulating UART2

When you now use the `printf()` from the `stdio.h` library to send text through the UART2 peripheral, that text will be sent to UART2 Output tab (usually at the bottom of the IDE) after compilation as in Figure B-7. You learn more about the `printf()` function in the early chapters of this manual. If you right click inside the window, you can clear the window, change the size and colour of the text, and print the output to a file. Press the *Finish* icon (red square) to stop the simulation. It will keep running because of the `while(1)` line which means wait forever.

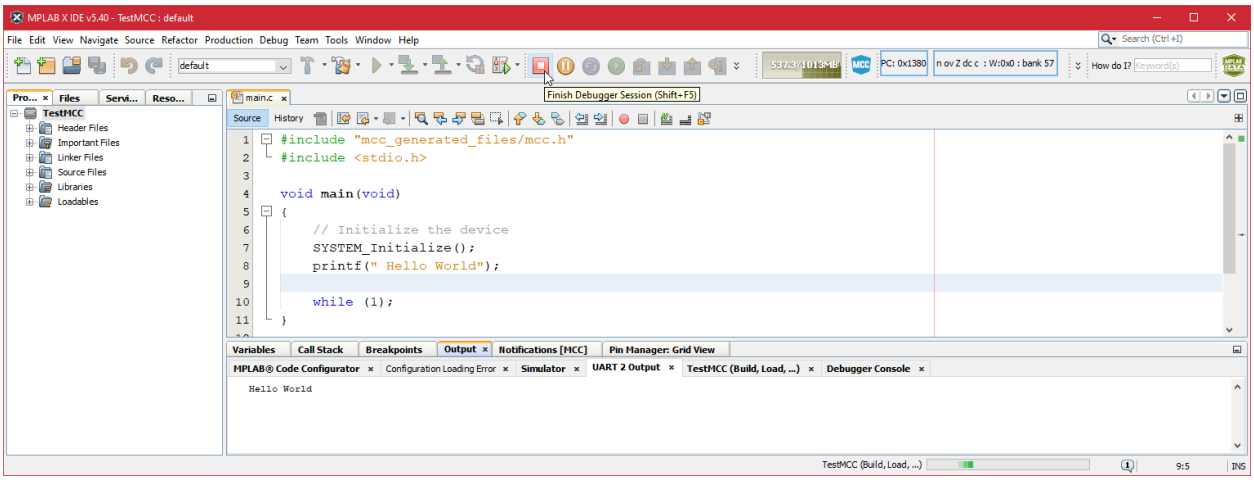



**Figure B-7:** UART2 window.

## Breakpoints

The simulator is particularly helpful for debugging when breakpoints are established. These allow you to monitor the status of the variables in your program at different stages of your program.

When breakpoints are set, the Simulator will execute the code but stop at the first breakpoint. If you also have a "watch window" open, you can see how variables and Special Function Registers change from breakpoint to breakpoint to breakpoint in the execution of your code. You move to the next breakpoint by clicking on the green arrow icon. All the intervening code is executed,

You set breakpoints in your code by right clicking on the number of the line in a code window where you wish to set the breakpoint. A small red square replaces the line number as in Figure B-9. Sometimes breakpoints cannot be set exactly where you want them to be. If that happens, insert a `NOP()`, a "do nothing" or "no operation" function, and set the break point on the line with the `NOP()`.

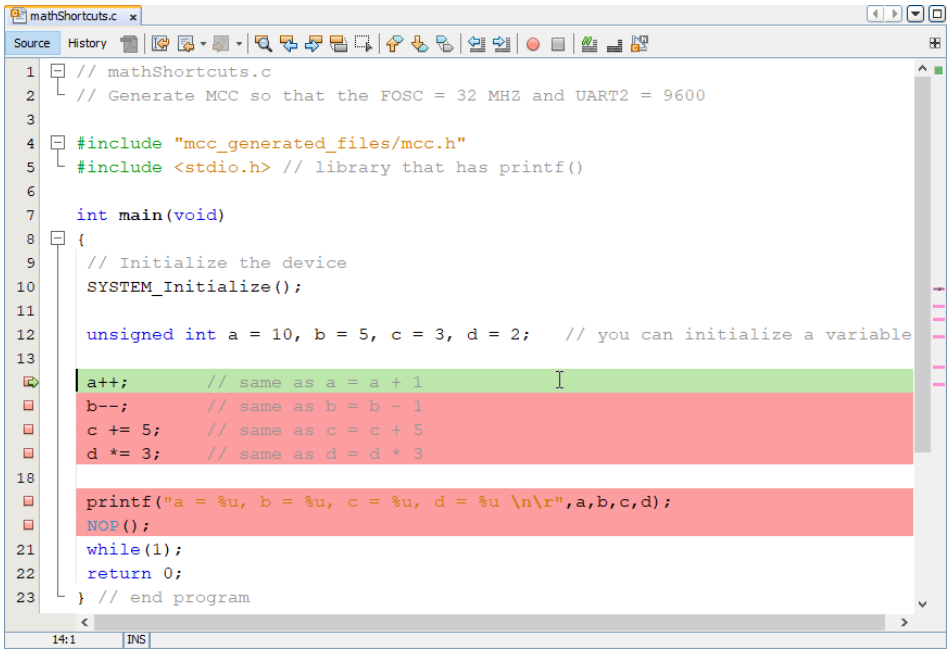



**Figure B-8:** Setting a breakpoint.

## Watching Variables

As you step through your code you can watch how your variables change from line to line in a "watch window". This is an excellent way to spot unexpected behaviour. You open a watch window from *Window|Debugging|Watches* as shown in Figure B-10. This will open a watch window as in Figure B-11. You can select any variable or Special Function Register (SFR) in your code to watch by right-clicking on the *<Enter new watch>* line in that window. This will bring up the *New Watch window,* Figure B-12. An SFR is an internal variable of a PIC/MCU peripheral and SFRs are discussed elsewhere in this manual. Variables are displayed in decimal, binary, and hexadecimal formats as well as type. Right-clicking on the top row of the Watch Window allows you to decide which characteristics to display as shown in Figure B-13. The variables will be "out of scope" or undefined before the program runs unless they have been initialized. If a breakpoint is placed on a line in main, and program execution stops on that line, the local variables will be in scope and will be displayed. Figure B-14 shows you the new values of the variables after stepping through the breakpoints.

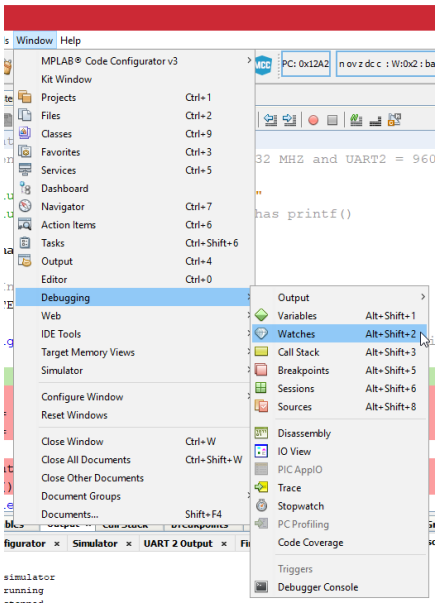



**Figure B-9:** Setting a Watch.

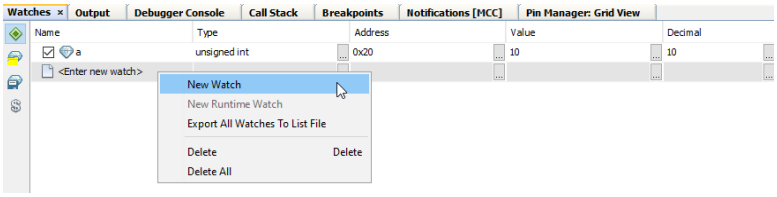



**Figure B-10:** Selecting the variables to watch.

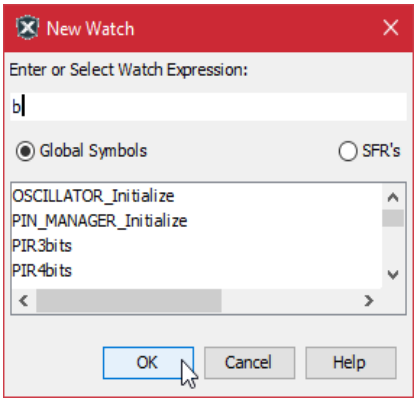



**Figure B-11:** Enter variable or SFR name.

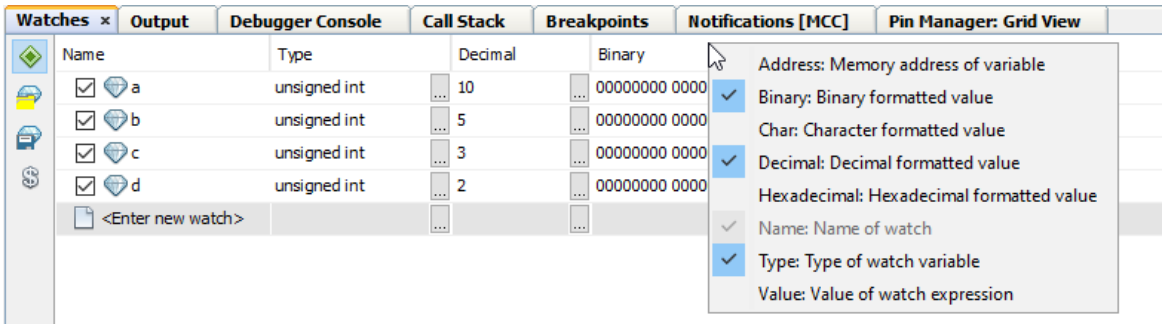



**Figure B-12:** Information to display.

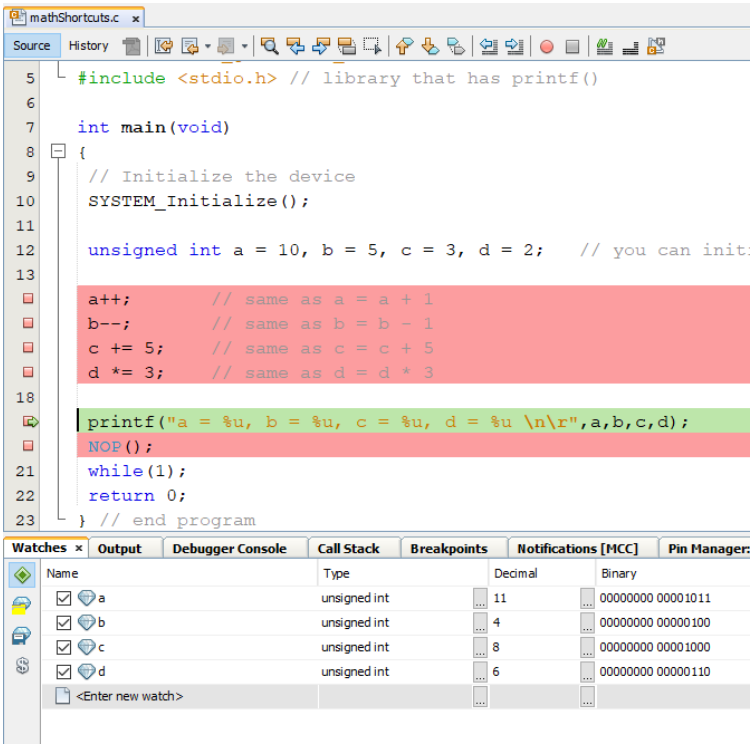



**Figure B-13:** Variable values at a new line.

## Stopwatch

Timing is an important part of MCU operations. In real applications, when things happen – such as a signal turning on or off – is critical. In the Simulator, you can examine how much time is required for each line, or set of lines, of code to execute. You call up a "stopwatch window" by *Window|Debugging|Stopwatch* as shown in Figure B-10. A stopwatch window is shown in Figure B-15. The lower three icons on the left side of the window are very useful. The trashcan icon will clear the window. The bottom stopwatch icons let you zero the stopwatch. The top stopwatch, resets or zeroes the stopwatch between breakpoints so that you can measure the time between steps as in Figure B-16. Note that the simulator must be set to the correct frequency of your PIC/MCU to yield the correct times.

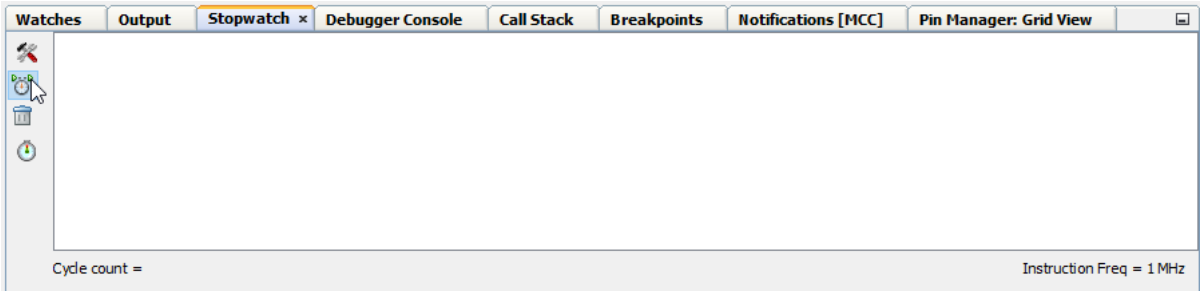



**Figure B-14:** Stopwatch window.

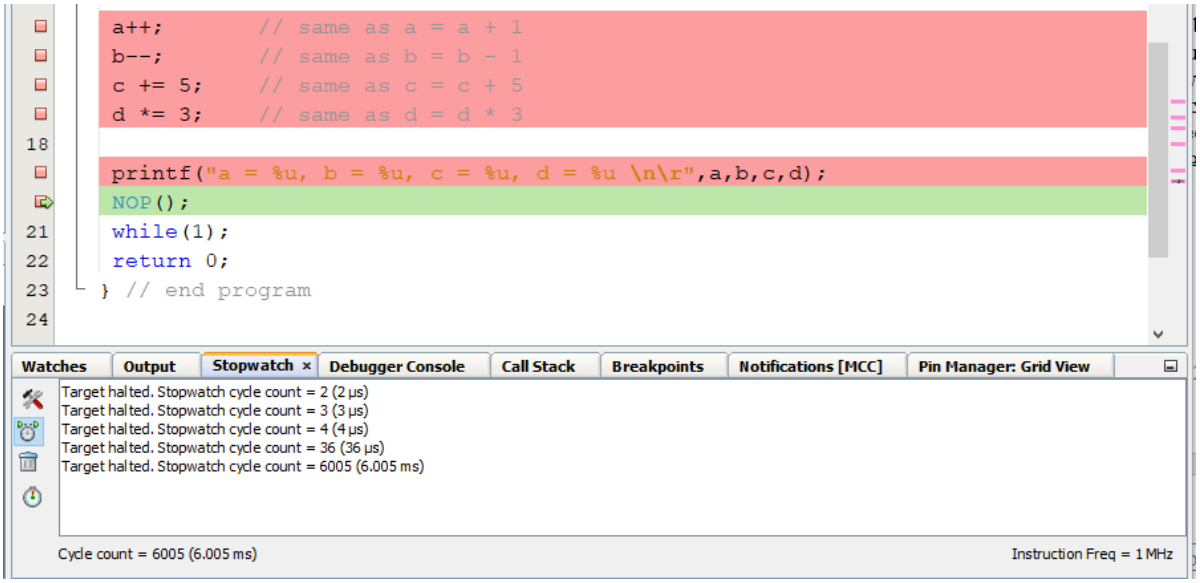



**Figure B-15:** Stopwatch cycle count between breaks.

# Appendix C. PuTTY

Most PCs no longer have their own COM ports which are the hardware used with UART or Serial communication. Virtual serial COM ports operating over a USB connection are used instead. UART peripherals cannot tell the difference between real and virtual COM ports. The MPLAB Xpress Board, has a USB serial COM port function that we will use.

When you first connect the Xpress Board to you PC, your PC will recognize it as a USB memory device and assign a Drive letter. The PC will also assign a name to the virtual COM port such as COM1 or COM3. To find the number of the virtual COM port, open *Control Panel:Device Manager* and expand *Ports (COM & LPT)*. You should see a line for the USB Serial Device (COM#) where # is the number assigned to the device as in Figure F-1 below.

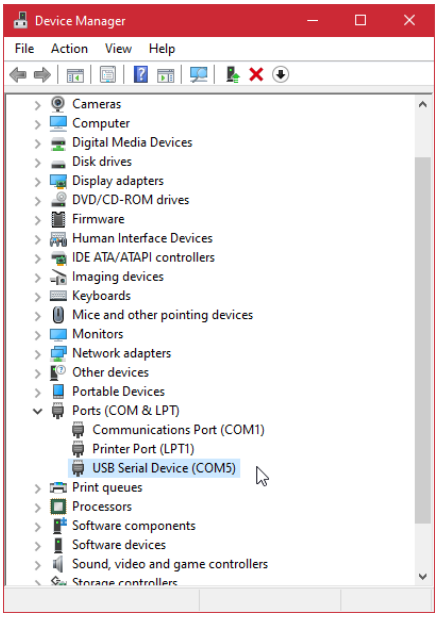



**Figure C-1:** Find your COM port in the Device Manager.

## Configuring PuTTY on the PC

On the PC side, we need a so-called terminal program to communicate with the PIC/MCU. We will use the free program PuTTY. PuTTY allows for communication via many protocols but we are only in the serial communication mode. If you have not already installed PuTTY, you can find the installer on the Moodle site for this course.

Open PuTTY. When PuTTY opens, see Figure F-2 , the Configuration Window appears. On the left-hand side, there is a frame called *Category*. Go to the very bottom and click on *Serial* under *Connection*.

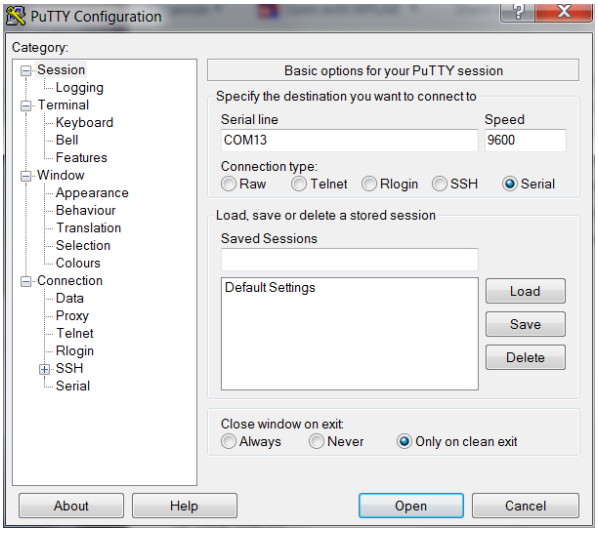



**Figure C-2: PuTTY Configuration.**

The window will change as shown in Figure F-3. On the right hand side, change the COM port to the one you found in Device Manager as noted above in Figure F-1. Change *Flow Control* at the bottom to *None*. Set the *Speed* to 9600, this is relatively slow but fine for keyboard use. You may change it to higher speeds later in your MPLAB projects if you wish.

If you have trouble with PuttY at higher speeds, try changing *Flow Control* to XON/XOFF.

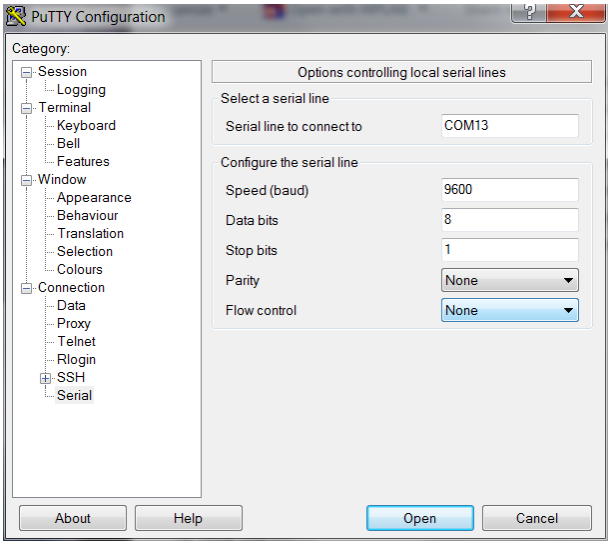



**Figure C-3:** Connection settings.

Next choose *Terminal* in the *Category* frame. The window will change to Figure F-4. You will need to select *Force On* for *Local Echoing*. That just means that you will see what you type in the terminal window.

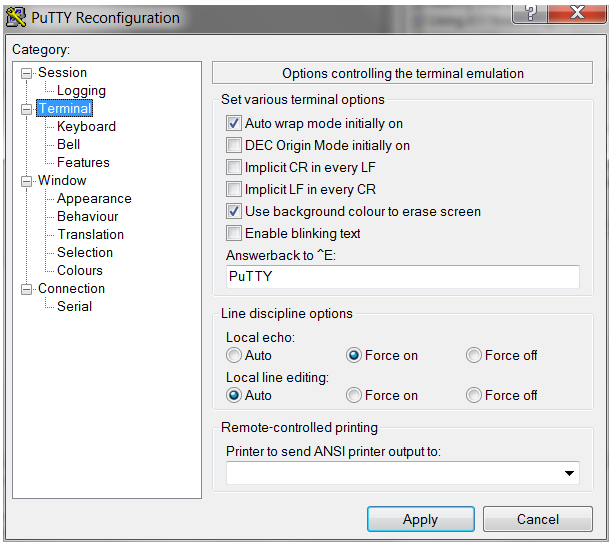



**Figure C-4:** Force local echoing.

Click on *Session* in the *Category* frame. This brings us back to Figure F-2. Here set the *Connection type* to *Serial*. For later, as a shortcut, under *Saved Sessions* put in the name of your COM port and then select *Save*. If you reopen PuTTY later, you don't need to reenter all these steps; just *Load* this session. Then click on *Open* to bring up the terminal window as shown in Figure F-5 below. Just click the mouse on the green cursor and begin typing.

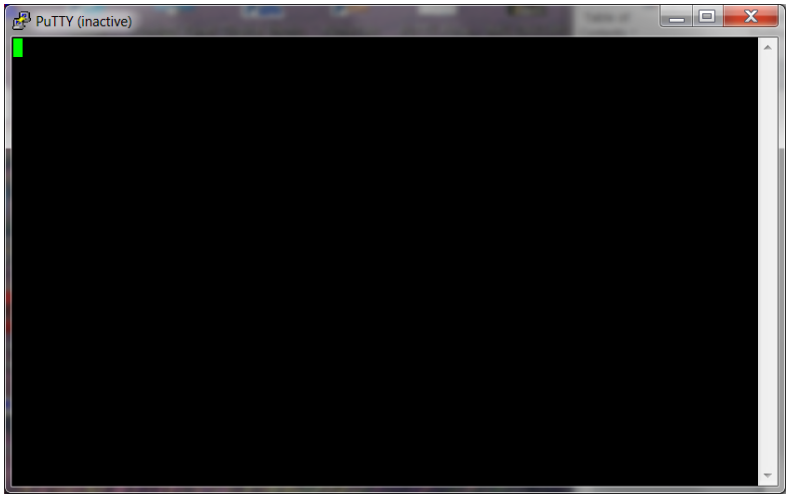



**Figure C-5: The PuTTY terminal window.**

Under *Window* in *Category*, you can find options to change the background to white, change the cursor style, and change the font, the font size, and the font colour. I use a white background and a larger font for the videos I create.

## Controlling the PuTTY Terminal Programmatically

You can change the PuTTY terminal window from program you write using what are known as VT-100 Escape sequences. The commands are of the form ESC[ followed by several more characters. Of especial use are the commands to clear the window and move the cursor to the top left corner shown below

```
printf("%c%c%c%c",27,'[','2','J'); // Clear PuTTY Window
printf("%c%c%c",27,'[','H');       // Home PuTTY Cursor
```

## Simple Serial Terminal

While PuTTY is quite versatile, MPLAB has an adequate terminal program plug-in. If you have installed it following the instructions in *Appendix A Software,* you open Simple Serial Port from *Tools|Embedded*.

When opened, Simple Serial Port should have its own tab at the bottom of the IDE. Along the bottom of the tab window, the status *Connected* or *Disconnected* and the COM port settings are displayed as shown in Figure F-6. Along the left side of the tab window are icons. The top question mark icon brings up a Help screen. The second icon, the crossed tools, is the settings menu where you can configure the COM port. The eraser-shaped icon lets you clear the window. The bottom icon that looks like a plug is where you connect or disconnect the COM port.

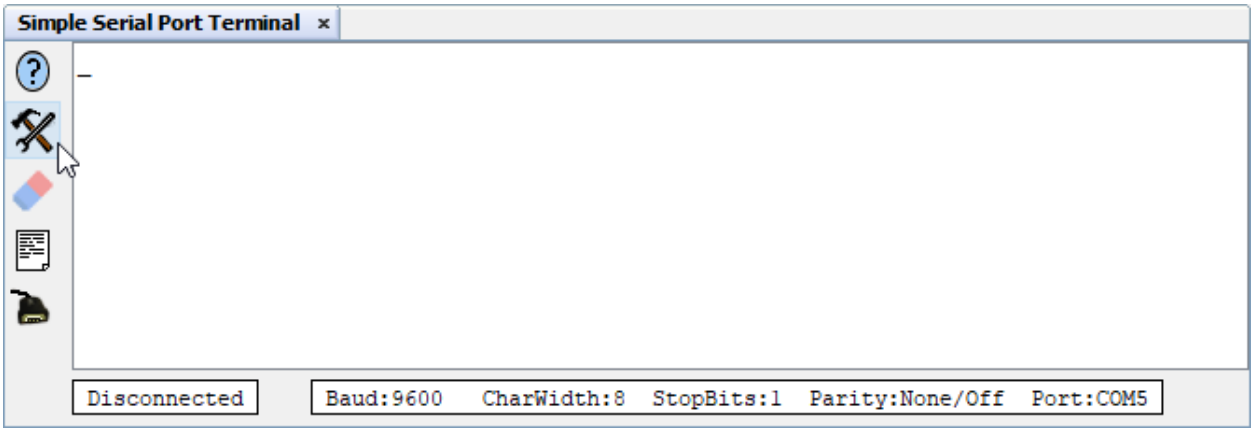



**Figure C-6:** The Simple Serial Port Terminal

Clicking on the Settings icon brings up the window shown in Figure F-7. Choose the same settings as PuTTY. Again, if there are problems at high speed, try changing *Flow Control* to *XON/XOFF*. You can create and save a settings file at the bottom of the window.

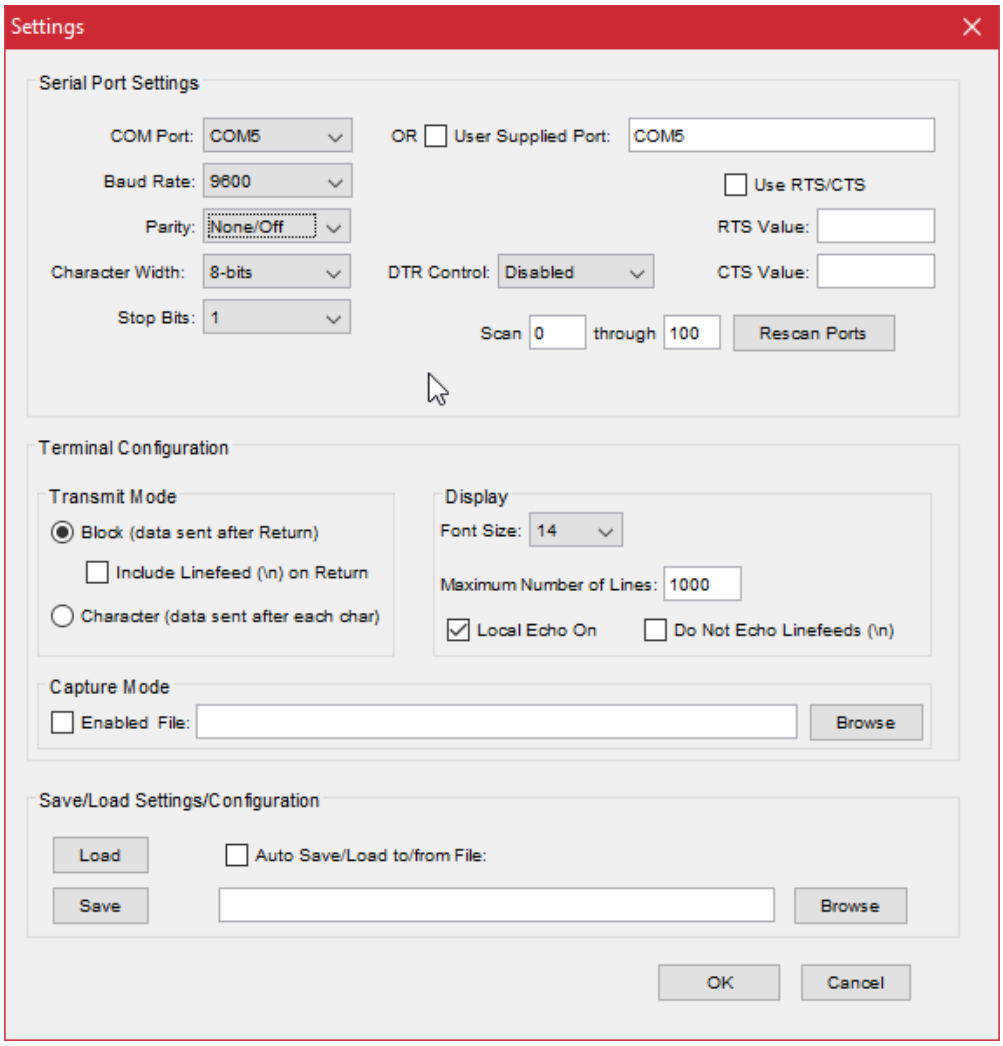



**Figure C-7:** COM port settings.

Please note that only one program on the PC can use the COM Port at a time.

## Data Visualizer

Data Visualizer is also a serial port terminal program but with a difference. It interprets numerical data as commands to move vertically up or down the screen and places a marker at the appropriate height. With each new data set, it moves the previous points to the right, creating a moving strip chart. The faster the data is received, the faster the datapoints move right. The datapoints can be joined by curves; either smooth point to point or stepwise. You open Data Visualizer from *Tools|Embedded* of from the icon on the Icon bar.

On the left top menu bar, Figure F-8, you can click on *Documentation* to open a manual on how to use Data Visualizer. On the left side, you can configure the connection settings of a selected

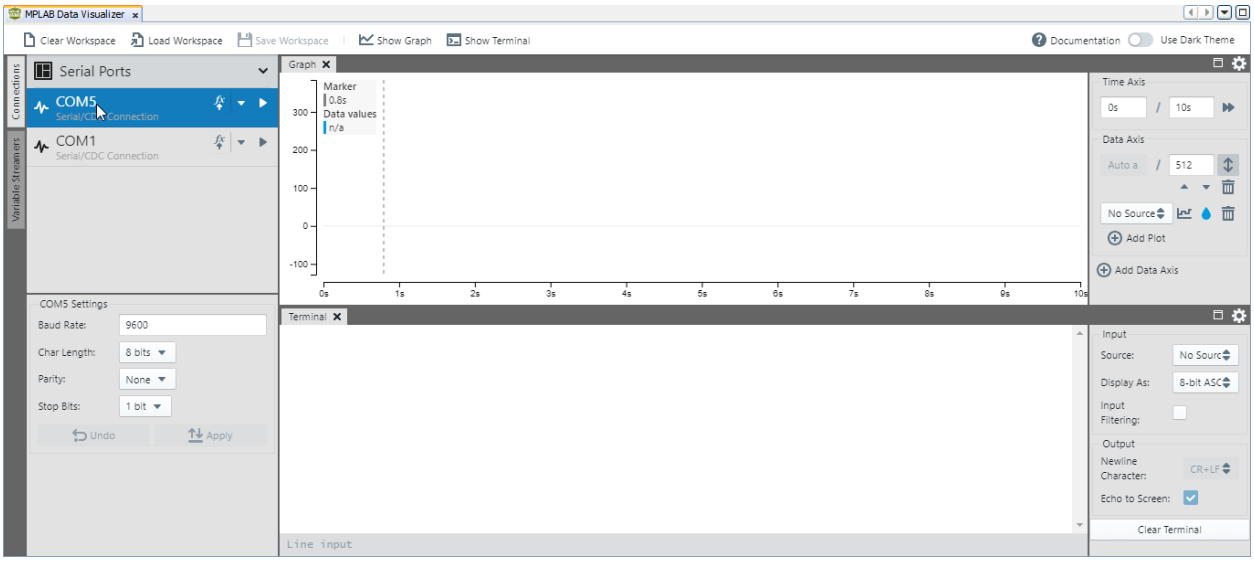



**Figure C-8:** Data Visualizer

COM port. The default is the standard 9600-8-none-1. The centre of the program screen has a top and bottom. The graph is displayed on the top and terminal data on the bottom. We do not need the terminal screen so x it off. Data needs the incoming data in a special form; it calls the form a *data stream*. You can define the data stream by clicking on the sidebar tab *Variable streamers* and clicking on *New Variable Streamer*. This brings up the window in Figure F-9. Here you name the graph and label one or more sets of data. You need to tell Data Visualizer the size and type of each variable. For simplicity stick to 8-bit and 16-bit integers, either signed or unsigned. When you click *Next*, a new window, Figure F-10, will ask how you want to plot the data. For example, each set on separate stacked axes as selected or on the plots on the same axes (the third choice). When you click on *Finish*, Data Visualizer will look like Figure F-11below. Also in Figure F-11, notice the little toolbar that appears when you hover the mouse above the top of the graph. It can ne very useful.

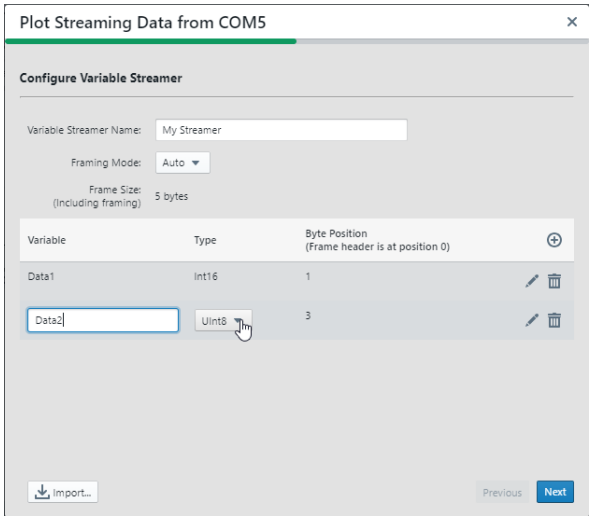



**Figure C-10:** Stream setup

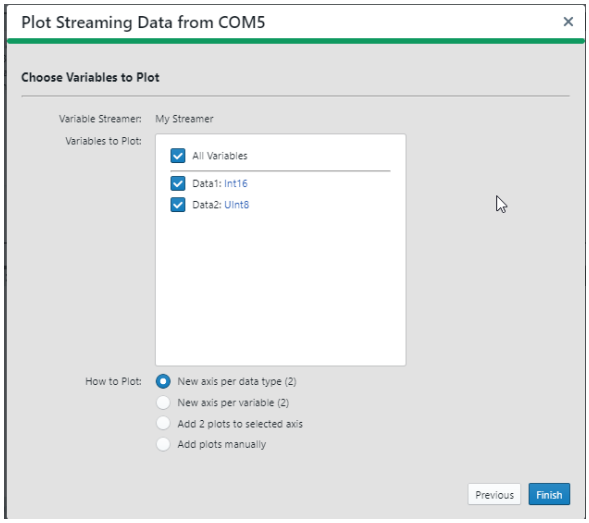



**Figure C-9:** Plot setup.

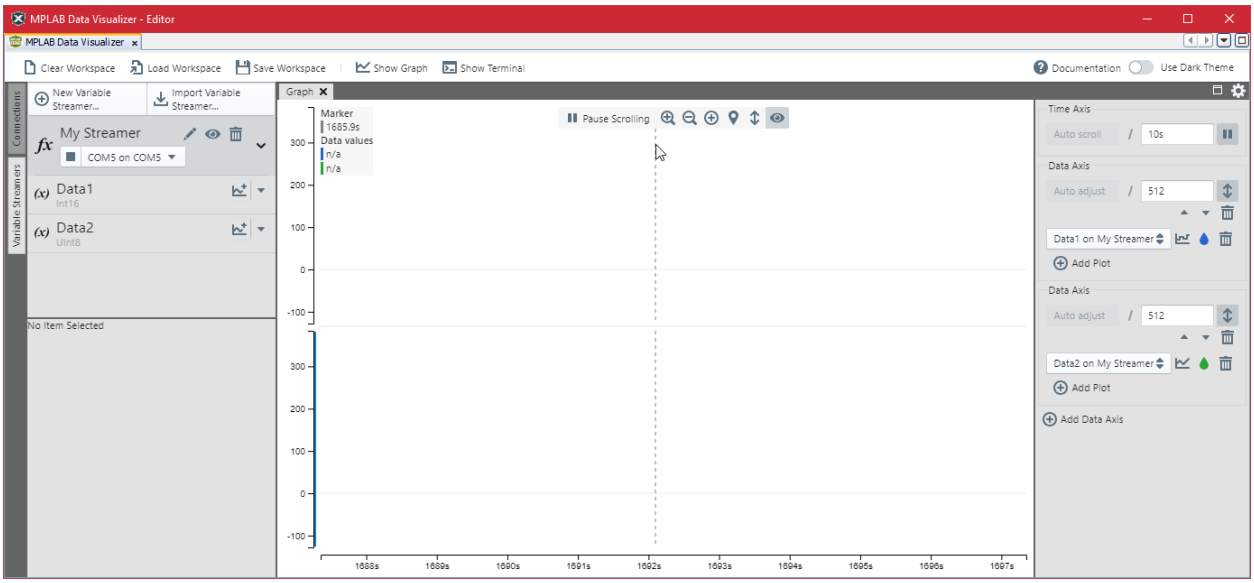



**Figure C-11:** Data Vis with streams.

The last point to make is how you must print out your data for Data Visualizer to interpret it into the appropriate data streams. You need to write your data to Data Visualizer in byte-sized chunks. For this reason in example program `DataVisTest.c` below, we need to use `UART2_Write(char letter)` instead of `printf()`. And the 2-byte, 16-bit, integer must be broken into its high byte and its low byte in reverse order. This is easier to understand in hexadecimal or base 16. Consider 0x7AF1 which is the same as decimal 31,473. The high byte is 0x7A or decimal 122 and the low byte is 0xF1 or decimal 241. Note $126 \times 256 + 241 = 31{,}473$. In the code below, `Data1 && 0x00FF` ANDs the upper byte with zeros and the lower byte with ones which make them yields 0x00F1 in our example. The line `Data1 >> 8` shifts the upper bits over into the lower bits position, so our example becomes 0x007A. `UART_Write()` ignores the upper byte when printing. Lastly, the data is bracketed by a frame start byte and a frame end byte, so that Data Visualizer understands where the data packet begins and ends. The result of the example code is shown in Figure F-12..

```
// DataVisTest.c
// system at 1 MHz
// UART2 at 9,600 bps

#include "mcc_generated_files/mcc.h"
#include <math.h>
#include <stdio.h>

#define _PI_  4.0*atan(1.0)

void main(void)
{
    // Initialize the device
    SYSTEM_Initialize();

    int i = 0 , Data1;  // signed 16-bit integer
    unsigned char Data2; // unsigned 8-bit int
```

```
    printf("Data Vis Test\n\r");
    IO_RD2_SetHigh();
    DELAY_milliseconds(200);
    IO_RD2_SetLow();
    DELAY_milliseconds(200);

    while (1)
    {
        // create some data to plot
        Data1 = (int)(1000.0*sin(_PI_ * i/10.0));
        Data2 = (unsigned char)(100.0*(cos(_PI_ * i/10.0) + 1.0));
        i++;
        if (i == 100) i = 0;  // reset i

        UART2_Write(0x5F);           // frame start Decimal 95

        // unsigned int value as two 8bit pieces to Data Visualizer
        UART2_Write(Data1 & 0x00FF); // First data set,
                                     // send lower 8 bits first
        UART2_Write(Data1 >> 8);     // send upper 8 bits second
        UART2_Write(Data2);          // Second data set

        UART2_Write(0xA0);           // frame end = 160 (Note 160 + 95 = 255)

        DELAY_milliseconds(50);
    }
}
```

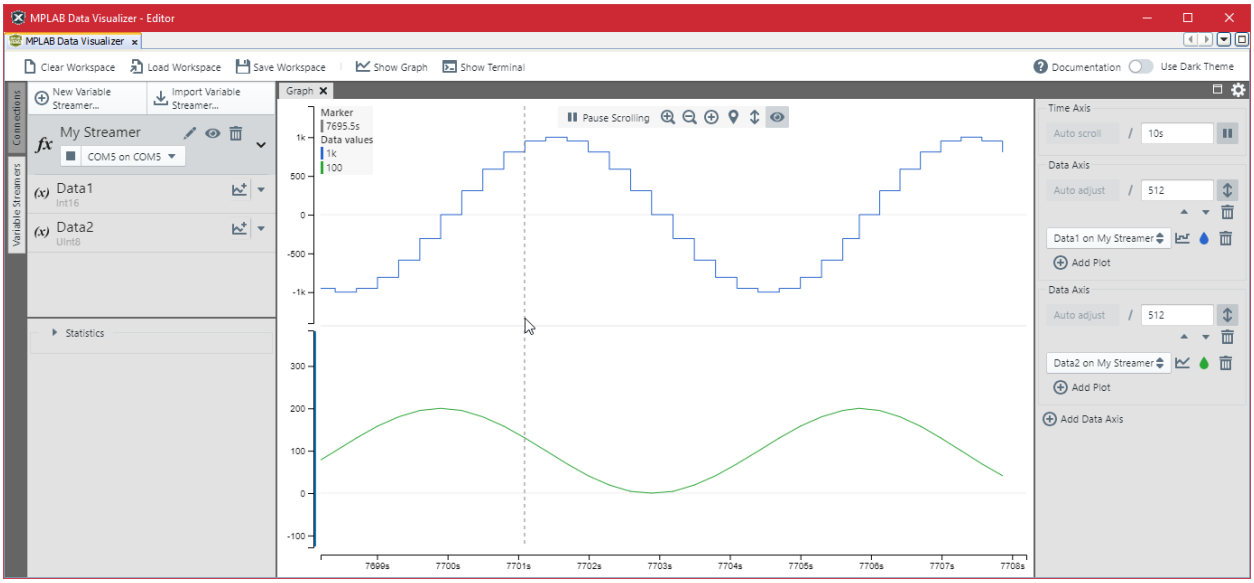



**Figure C-12:** Data Visualizer and data

# Appendix D. ASCII

The char data type in the C language is just a number. However, every possible value of a char number is associated with a text symbol or special function for printing, such as a carriage return, as shown in Table B-1 below. The special non-printing characters are shown in bold in the table below and discussed in more detail in the next section. When characters are printed, e.g. using the `printf()` function, the number is replaced by the symbol. Which symbol replaces which number, depends on an arbitrary convention. The particular convention used in the table below is called ASCII. Other conventions, and modifications of the ASCII standard convention, exist – especially for different languages. As well, there is the Extended ASCII Character set for decimal values from 127 to 255. These can be very different from device to device such as the SmartFun LCD and a PC.

**Table D-1:** The ASCII character codes.

| Char | Dec | Oct | Hex | Bin | | Char | Dec | Oct | Hex | Bin |
|---|---|---|---|---|---|---|---|---|---|---|
| (nul) | 0 | 0 | 0x00 | 0b00000000 | | **(us)** | **31** | **37** | **0x1f** | **0b00011111** |
| (soh) | 1 | 1 | 0x01 | 0b00000001 | | (sp) | 32 | 40 | 0x20 | 0b00100000 |
| (stx) | 2 | 2 | 0x02 | 0b00000010 | | ! | 33 | 41 | 0x21 | 0b00100001 |
| (etx) | 3 | 3 | 0x03 | 0b00000011 | | " | 34 | 42 | 0x22 | 0b00100010 |
| (eot) | 4 | 4 | 0x04 | 0b00000100 | | # | 35 | 43 | 0x23 | 0b00100011 |
| (enq) | 5 | 5 | 0x05 | 0b00000101 | | $ | 36 | 44 | 0x24 | 0b00100100 |
| (ack) | 6 | 6 | 0x06 | 0b00000110 | | % | 37 | 45 | 0x25 | 0b00100101 |
| (bel) | 7 | 7 | 0x07 | 0b00000111 | | & | 38 | 46 | 0x26 | 0b00100110 |
| (bs) | 8 | 10 | 0x08 | 0b00001000 | | ' | 39 | 47 | 0x27 | 0b00100111 |
| (ht) | 9 | 11 | 0x09 | 0b00001001 | | ( | 40 | 50 | 0x28 | 0b00101000 |
| (nl) | 10 | 12 | 0x0a | 0b00001010 | | ) | 41 | 51 | 0x29 | 0b00101001 |
| (vt) | 11 | 13 | 0x0b | 0b00001011 | | * | 42 | 52 | 0x2a | 0b00101010 |
| (np) | 12 | 14 | 0x0c | 0b00001100 | | + | 43 | 53 | 0x2b | 0b00101011 |
| (cr) | 13 | 15 | 0x0d | 0b00001101 | | , | 44 | 54 | 0x2c | 0b00101100 |
| (so) | 14 | 16 | 0x0e | 0b00001110 | | - | 45 | 55 | 0x2d | 0b00101101 |
| (si) | 15 | 17 | 0x0f | 0b00001111 | | . | 46 | 56 | 0x2e | 0b00101110 |
| (dle) | 16 | 20 | 0x10 | 0b00010000 | | / | 47 | 57 | 0x2f | 0b00101111 |
| (dc1) | 17 | 21 | 0x11 | 0b00010001 | | 0 | 48 | 60 | 0x30 | 0b00110000 |
| (dc2) | 18 | 22 | 0x12 | 0b00010010 | | 1 | 49 | 61 | 0x31 | 0b00110001 |
| (dc3) | 19 | 23 | 0x13 | 0b00010011 | | 2 | 50 | 62 | 0x32 | 0b00110010 |
| (dc4) | 20 | 24 | 0x14 | 0b00010100 | | 3 | 51 | 63 | 0x33 | 0b00110011 |
| (nak) | 21 | 25 | 0x15 | 0b00010101 | | 4 | 52 | 64 | 0x34 | 0b00110100 |
| (syn) | 22 | 26 | 0x16 | 0b00010110 | | 5 | 53 | 65 | 0x35 | 0b00110101 |
| (etb) | 23 | 27 | 0x17 | 0b00010111 | | 6 | 54 | 66 | 0x36 | 0b00110110 |
| (can) | 24 | 30 | 0x18 | 0b00011000 | | 7 | 55 | 67 | 0x37 | 0b00110111 |
| (em) | 25 | 31 | 0x19 | 0b00011001 | | 8 | 56 | 70 | 0x38 | 0b00111000 |
| (sub) | 26 | 32 | 0x1a | 0b00011010 | | 9 | 57 | 71 | 0x39 | 0b00111001 |
| (esc) | 27 | 33 | 0x1b | 0b00011011 | | : | 58 | 72 | 0x3a | 0b00111010 |
| (fs) | 28 | 34 | 0x1c | 0b00011100 | | ; | 59 | 73 | 0x3b | 0b00111011 |
| (gs) | 29 | 35 | 0x1d | 0b00011101 | | < | 60 | 74 | 0x3c | 0b00111100 |
| **(rs)** | **30** | **36** | **0x1e** | **0b00011110** | | = | 61 | 75 | 0x3d | 0b00111101 |
| **Char** | **Dec** | **Oct** | **Hex** | **Bin** | | **Char** | **Dec** | **Oct** | **Hex** | **Bin** |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| > | 62 | 76 | 0x3e | 0b00111110 | | _ | 95 | 137 | 0x5f | 0b01011111 |
| ? | 63 | 77 | 0x3f | 0b00111111 | | ` | 96 | 140 | 0x60 | 0b01100000 |
| @ | 64 | 100 | 0x40 | 0b01000000 | | a | 97 | 141 | 0x61 | 0b01100001 |
| A | 65 | 101 | 0x41 | 0b01000001 | | b | 98 | 142 | 0x62 | 0b01100010 |
| B | 66 | 102 | 0x42 | 0b01000010 | | c | 99 | 143 | 0x63 | 0b01100011 |
| C | 67 | 103 | 0x43 | 0b01000011 | | d | 100 | 144 | 0x64 | 0b01100100 |
| D | 68 | 104 | 0x44 | 0b01000100 | | e | 101 | 145 | 0x65 | 0b01100101 |
| E | 69 | 105 | 0x45 | 0b01000101 | | f | 102 | 146 | 0x66 | 0b01100110 |
| F | 70 | 106 | 0x46 | 0b01000110 | | g | 103 | 147 | 0x67 | 0b01100111 |
| G | 71 | 107 | 0x47 | 0b01000111 | | h | 104 | 150 | 0x68 | 0b01101000 |
| H | 72 | 110 | 0x48 | 0b01001000 | | i | 105 | 151 | 0x69 | 0b01101001 |
| I | 73 | 111 | 0x49 | 0b01001001 | | j | 106 | 152 | 0x6a | 0b01101010 |
| J | 74 | 112 | 0x4a | 0b01001010 | | k | 107 | 153 | 0x6b | 0b01101011 |
| K | 75 | 113 | 0x4b | 0b01001011 | | l | 108 | 154 | 0x6c | 0b01101100 |
| L | 76 | 114 | 0x4c | 0b01001100 | | m | 109 | 155 | 0x6d | 0b01101101 |
| M | 77 | 115 | 0x4d | 0b01001101 | | n | 110 | 156 | 0x6e | 0b01101110 |
| N | 78 | 116 | 0x4e | 0b01001110 | | o | 111 | 157 | 0x6f | 0b01101111 |
| O | 79 | 117 | 0x4f | 0b01001111 | | p | 112 | 160 | 0x70 | 0b01110000 |
| P | 80 | 120 | 0x50 | 0b01010000 | | q | 113 | 161 | 0x71 | 0b01110001 |
| Q | 81 | 121 | 0x51 | 0b01010001 | | r | 114 | 162 | 0x72 | 0b01110010 |
| R | 82 | 122 | 0x52 | 0b01010010 | | s | 115 | 163 | 0x73 | 0b01110011 |
| S | 83 | 123 | 0x53 | 0b01010011 | | t | 116 | 164 | 0x74 | 0b01110100 |
| T | 84 | 124 | 0x54 | 0b01010100 | | u | 117 | 165 | 0x75 | 0b01110101 |
| U | 85 | 125 | 0x55 | 0b01010101 | | v | 118 | 166 | 0x76 | 0b01110110 |
| V | 86 | 126 | 0x56 | 0b01010110 | | w | 119 | 167 | 0x77 | 0b01110111 |
| W | 87 | 127 | 0x57 | 0b01010111 | | x | 120 | 170 | 0x78 | 0b01111000 |
| X | 88 | 130 | 0x58 | 0b01011000 | | y | 121 | 171 | 0x79 | 0b01111001 |
| Y | 89 | 131 | 0x59 | 0b01011001 | | z | 122 | 172 | 0x7a | 0b01111010 |
| Z | 90 | 132 | 0x5a | 0b01011010 | | { | 123 | 173 | 0x7b | 0b01111011 |
| [ | 91 | 133 | 0x5b | 0b01011011 | | \| | **124** | **174** | **0x7c** | **0b01111100** |
| \ | 92 | 134 | 0x5c | 0b01011100 | | } | 125 | 175 | 0x7d | 0b01111101 |
| ] | 93 | 135 | 0x5d | 0b01011101 | | ~ | 126 | 176 | 0x7e | 0b01111110 |
| ^ | 94 | 136 | 0x5e | 0b01011110 | | (del) | 127 | 177 | 0x7f | 0b01111111 |

## ASCII Control Characters

Some devices, such as the SparkFun LCD, use the non-printing ASCII characters 0 to 31 as control characters or commands. The control character which is number 10 in the ASCII table is often denoted <control>M or CTRL-M or ^M in documents with similar notation for the other controls. Different devices may use the same control character to do different things. It is important to look up the device documentation to find the meaning of each control character. The SparkFun LCD documentation indicates that you can select a baud rate of 9600 by first entering special command character 124 followed by <control>M. Since <control>M means character 13, you would need the line of code `printf("%c%c", 124, 13);` in your program to make this selection. A list of the control characters and the common notation is given below. Note that character 124, the vertical

stroke |, is thus not available. SparkFun also makes use of extended ASCII character 254 as a control character.

**Table D-2:** ASCII Control Characters

| Dec | Hex | Command | | Dec | Hex | Comand | | Dec | Hex | Comand |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0x00 | CTRL-@ | | 11 | 0x0B | CTRL-K | | 22 | 0x16 | CTRL-V |
| 1 | 0x01 | CTRL-A | | 12 | 0x0C | CTRL-L | | 23 | 0x17 | CTRL-W |
| 2 | 0x02 | CTRL-B | | 13 | 0x0D | CTRL-M | | 24 | 0x18 | CTRL-X |
| 3 | 0x03 | CTRL-C | | 14 | 0x0E | CTRL-N | | 25 | 0x19 | CTRL-Y |
| 4 | 0x04 | CTRL-D | | 15 | 0x0F | CTRL-O | | 26 | 0x1A | CTRL-Z |
| 5 | 0x05 | CTRL-E | | 16 | 0x10 | CTRL-P | | 27 | 0x1B | CTRL-[ |
| 6 | 0x06 | CTRL-F | | 17 | 0x11 | CTRL-Q | | 28 | 0x1C | CTRL-\ |
| 7 | 0x07 | CTRL-G | | 18 | 0x12 | CTRL-R | | 29 | 0x1D | CTRL-] |
| 8 | 0x08 | CTRL-H | | 19 | 0x13 | CTRL-S | | 30 | 0x1E | CTRL-^ |
| 9 | 0x09 | CTRL-I | | 20 | 0x14 | CTRL-T | | 31 | 0x1F | CTRL-_ |
| 10 | 0x0A | CTRL-J | | 21 | 0x15 | CTRL-U | | | | |

# Appendix E. The Oscilloscope

The oscilloscope is a standard tool in laboratories and industrial settings wherever electronic equipment needs to be tested. Its inventor, K.F. Braun, a German physicist, shared the 1909 Nobel Prize with Marconi for their work in perfecting the radio. At its simplest, an oscilloscope is a voltmeter with a screen showing how a voltage signal changes with time. Most oscilloscopes have a bewildering number of dials, button, and switches. Fortunately, most can be ignored except for advanced uses. The controls you need to learn are indicated in the picture of the Tektonix 1000 Series Digital Oscilloscope shown in Figure D-1.

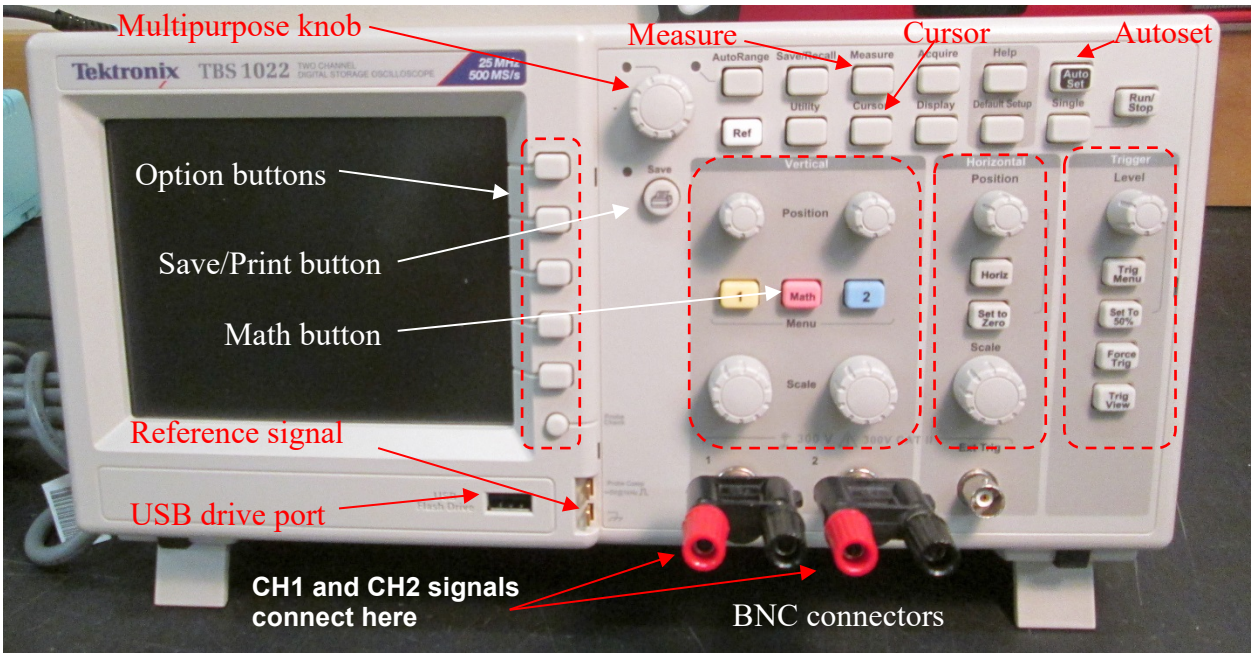



**Figure E-1:** Digital Oscilloscope

Notice that the oscilloscope controls are grouped into sections. The control groups we will concentrate on are the sections labelled *Vertical*, *Horizontal*, and *Trigger* in Figure D-1. There are also a set of *Option* buttons beside the display.

An important concept in the use of the oscilloscope is the *Trigger Condition*. An oscilloscope waits until the voltage either rises above, or falls below, a certain voltage and then commences to draw the signal starting on the left-hand side of the screen. The trigger condition is set by the the *Level* and *TRIG Menu* controls on the digital scope in Figure D-1. The *Sec/Div* dial, in the *Horizontal* controls group, controls how large a horizontal chunk, in seconds, of the signal the oscilloscope shows.

Figure D-2 illustrates how the trigger condition and the time setting control how a periodic triangular voltage signal would be displayed on the screen. Figure D-2 shows a trigger condition set to 2 Volts with a rising slope and a timing length *T*. Once the first trigger condition is met, the oscilloscope draws a portion of the signal of time length *T* on the screen. The oscilloscope then waits a small time delay $\Delta T$ until the trigger condition is met again and it then redraws the screen, and so on. If the timing length *T* is small enough, you don't notice the redrawing of the screen and your oscilloscope appears to have captured a picture of a segment of the voltage signal.

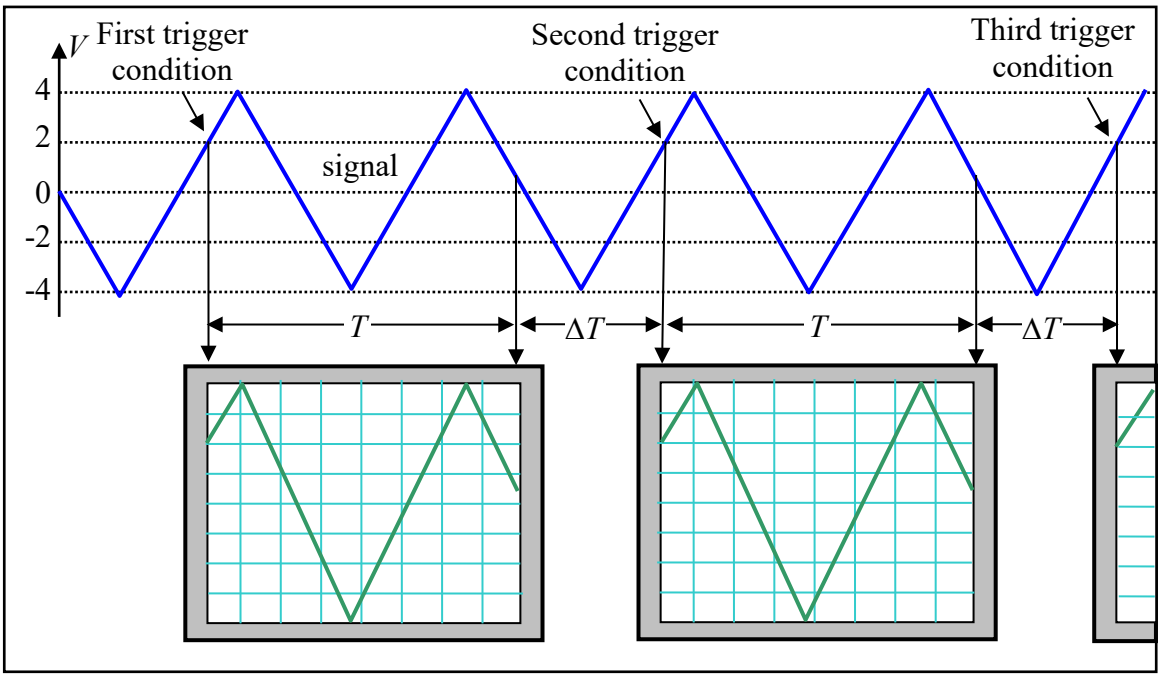



**Figure E-2:** How an oscilloscope displays a periodic voltage.

*What would the signal have looked like on the screen if the trigger condition had been 2 Volts and a falling slope?*

An important and useful feature of the oscilloscope is that it can display two voltage signals at once allowing all sorts of interesting comparisons. The first signal is yellow on the display and is called *CH 1* (Channel 1) and the second *CH 2* (Channel 2) is blue. The positioning controls (see Figure D-1) allow the signals to be moved around and even laid on top of one another. Pressing the buttons labelled [1] or [2] and using the choices presented by the *Options* buttons allows you to show *CH 1* only, *CH 2* only, or both together.

There is a complication with measuring two signals at once. Equipment plugged into wall outlets and handling large voltages and currents must be grounded. In the event of a malfunction, the ground provides a low resistance path for a current to travel. As currents prefer the path of least resistance, the high-resistance person using the equipment is safer. However this means that both channels must have a connection to ground. A typical arrangement might look like Figure D-3. The black lead from each of CH 1 and CH 2 would need to be connected to ground on the power

supply. When the red lead is connected to the circuit, the channel read the voltage between that point and ground. As a result, in this configuration, CH 1 measures $V_1 + V_2$ and CH 2 records $V_2$.

There is a neat trick that allows the oscilloscope to display $V_1$ for the circuit in Figure D-3. If you press the pink *MATH* button, the option buttons allow you to show a third signal in red, either Ch1 – Ch2, Ch1 + Ch2, or Ch1 × Ch2. Choosing Ch1 – Ch2 combines the signals from *CH1* and *CH2*, so in this case the oscilloscope would display $(V_1 + V_2) + (-V_2) = V_1$.

The timing length *T* displayed on the screen is determined from the number of horizontal squares times the setting on the *Sec/Div* dial which is the large knob in the centre of horizontal (time) controls. For example, if the screen has 8 squares and the dial is set to 0.1 ms, the total time displayed is 8 milliseconds. The maximum voltage is determined by one-half the number of vertical squares times the setting on the *Volts/Div* controls which are the large dials in the vertical

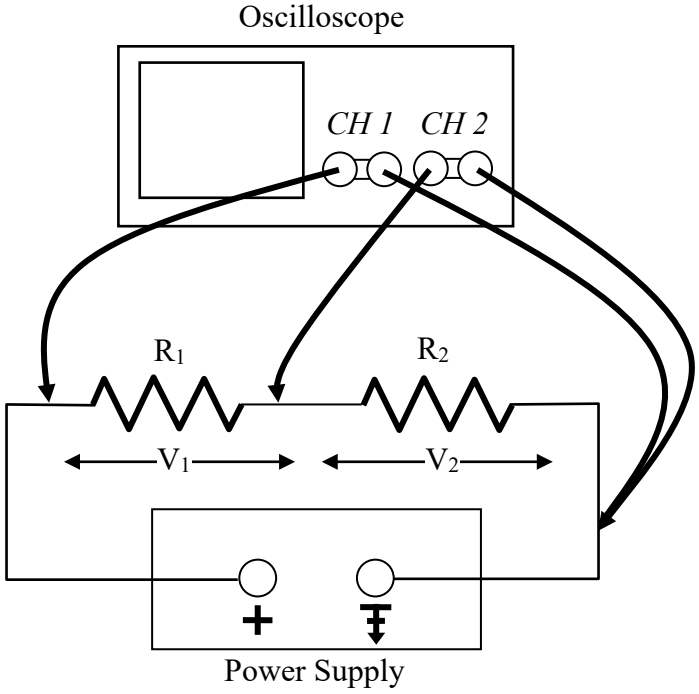



**Figure E-3:** How to connect an oscilloscope to read two voltages at once.

(voltage) controls section (see Figure D-1). The minimum voltage is the negative of the maximum value and zero volts is the vertical centre of the screen. For example on a screen with 8 vertical squares, called graticules, and the *CH 1 Volts/Div* dial set to 1 Volt, the top of the screen is 4 Volts and the bottom is -4 volts. There are two *Volts/Div* dials because the scale can be set for the signal from Channel 2 separately from Channel 1. Watch out for this when you make voltage measurements! Both signals have the same time setting though.

When you are examining a single channel, say by pressing [1], the Options buttons let's you to switch between *AC, GND,* and *DC* switch. Setting to GND (for ground) connects the red lead on the channel to the ground ("grounds out the signal") for the oscilloscope no matter where it is connected in a circuit. Since the oscilloscope and power supply are connected to the same building electrical system ground, the difference between the two ground connections is therefore zero

voltage and a horizontal line is displayed across the screen. This switch setting is useful is you have used the vertical positioning controls (see Figure D-1) to move the signal around; it allows you to find where zero volts is. You can use the vertical positioning control to move the zero line to a convenient location.

Some signals average out to zero over a full period and are said to be *pure AC* (alternating current). Some signals have a constant or *DC* (Direct Current) voltage. Other signals are a mixture of AC and DC. When the *AC, GND, and DC* choice is set to *DC*, the full signal is displayed on the screen. With the *AC* option chosen, the oscilloscope internally removes the constant DC portion of the voltage from the mixed signal, leaving a pure AC signal on the screen. Normally you leave the switch set to *DC*.

## Using the Oscilloscope

One nice feature of modern digital oscilloscopes is that they are smart enough, most of the time, to display the signals correctly. Simply press the *AUTOSET* button after you have you have attached the probes to your circuit. You can then adjust the voltage and timing and vertical and horizontal position to adjust the display to your needs.

The oscilloscope can also automatically make basic measurements of your signal such as peak-to-peak voltage and frequency. Press the *MEASURE* button and use the *OPTIONS* buttons to select which measurement to display.

If the type of measurement is not available from the basic measurements of if you want to make a measurement yourself, you can press the *CURSOR* button. Two vertical (time) or two horizontal (voltage) lines will appear on the display. The position of the lines on the display is controlled by you using the *MULTIPURPOSE* knob and the *OPTIONS* buttons. The display will give the value at the line's position of the difference between the two lines.

## Reference Voltage and Frequency

The digital oscilloscope has two pins on the front just below the LCD screen. These pins output a square wave, exactly 0 to 5 Volts at 1000 Hz and 50% duty cycle. The reference pins can be used to self-calibrate the scope. If you have programmed you MCU to measure voltage or time edges, these reference pins are a handy check to see if you MCU program is working correctly.

## USB Flash Drive

Insert a USB memory device into the socket. Press the *SAVE* button and choose among the *OPTIONS* buttons to do such things as save a picture of your display to the drive.

# Appendix F. Graphing with MS Excel

Frequently you will have a set of data which you wish to plot and examine for simple mathematical relationships. MS Excel lets you do this easily. First open Excel and enter your data in a column. Highlight the data and choose the Chart Wizard icon as shown in Figure E-1 below.

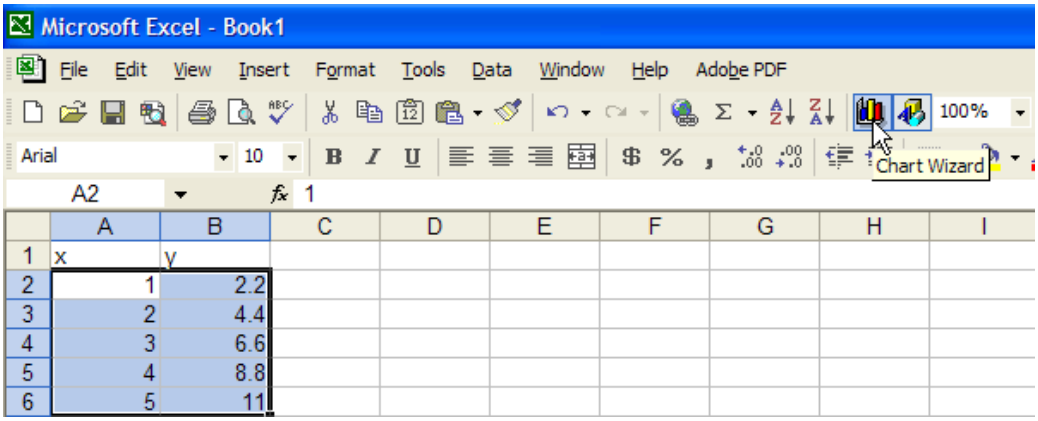



**Figure F-1:** Starting the Chart Wizard.

In the first window, Figure E-2, of the Chart Wizard choose the XY (Scatter) plot with no connecting lines.

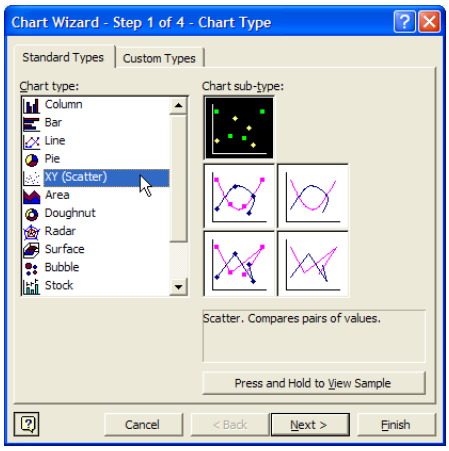



**Figure F-2:** Choosing an xy plot.

Continue to Step 3 of the Chart Wizard where you have multiple tabs with which to control the look of your plot including adding appropriate labels. See Figure E-3 a, b, & c.

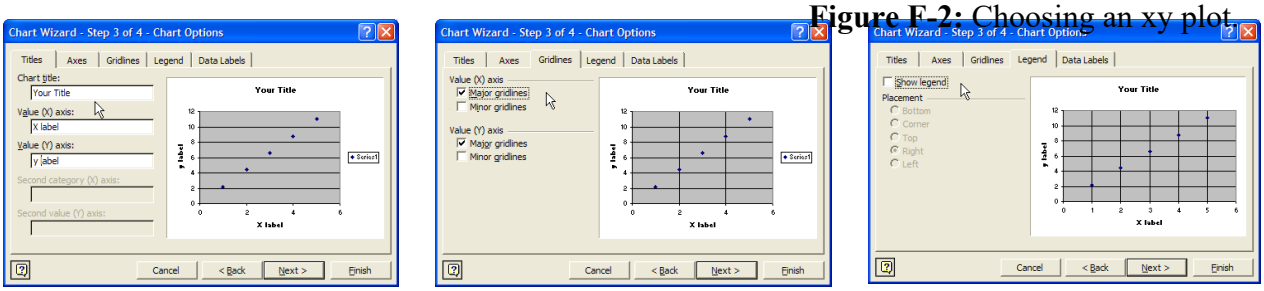



**Figure F-3 a, b, and c:** Controlling the look of the plot.

In Step 4 of the Chart Wizard, Figure E-4, you will be asked where to put the graph either in the sheet with the data or on a separate chart page of the workbook. It is usually a good idea to have it on its own sheet for editing purposes.

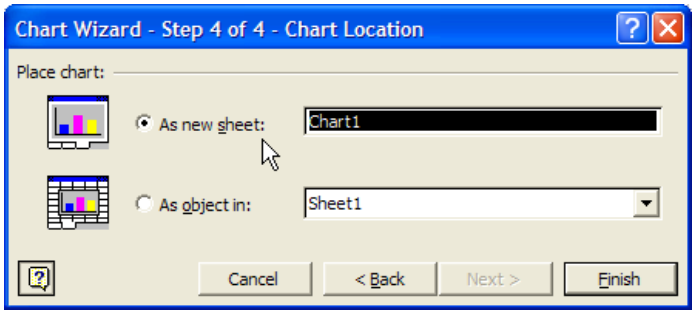



**Figure F-4:** Putting the plot on a new sheet.

When the chart appears, it will have a grey background. Right clicking on the background of the plot brings up a context window. If you select *Format Plot Area*, the new window lets you change the background. See Figure E-5.

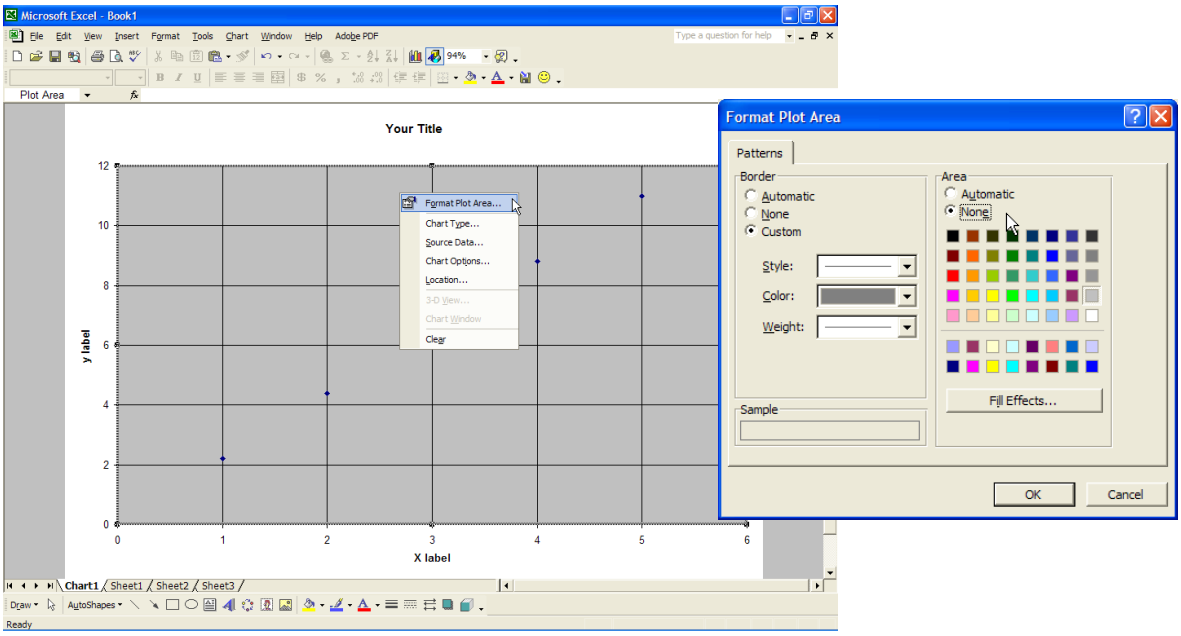



**Figure F-5 a & b.** Changing the plot area colour.

MS Excel calls the curve it fits to a set of points a trendline. To generate a trendline, highlight the data points by clicking on one of them. Next, right click to bring up a context window and choose *Add Trendline* as shown in Figure F-6a. The window that appears has two tabs. The first tab, Figure E-6b, lets you choose the sort of curve that Excel will fit to the data. You have lots of choice: Linear, Logarithmic, Polynomial (note a quadratic is a polynomial of order 2), Power, and Exponential. We are not interested in Moving Averages. In the second tab, shown in Figure E-6c, you can display the equation of the line and the R-squared value. The R-squared value is measure

of how well the data fits the trend line with 1 being a perfect fit. Try the different types of trendline until you get the best fit.

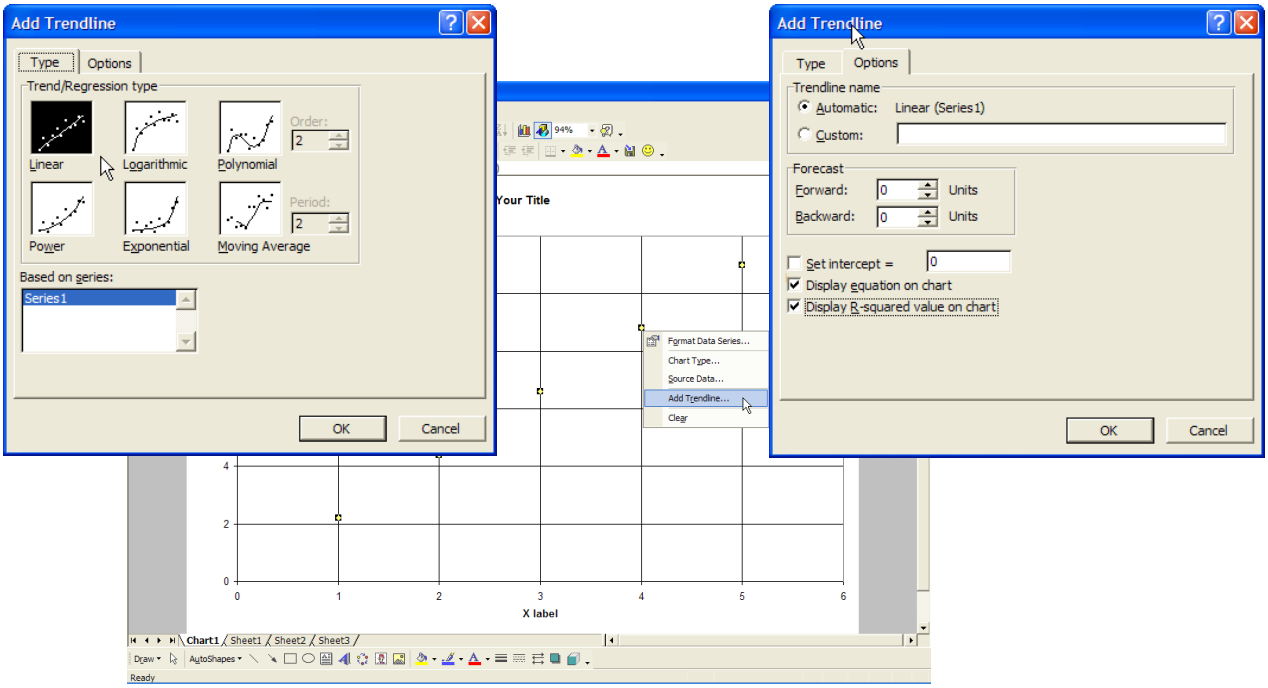



**Figure F-6 a, b, & c:** Fitting a trendline.

Figure E-7 shows the finished plot with the data, a trendline, and an equation.

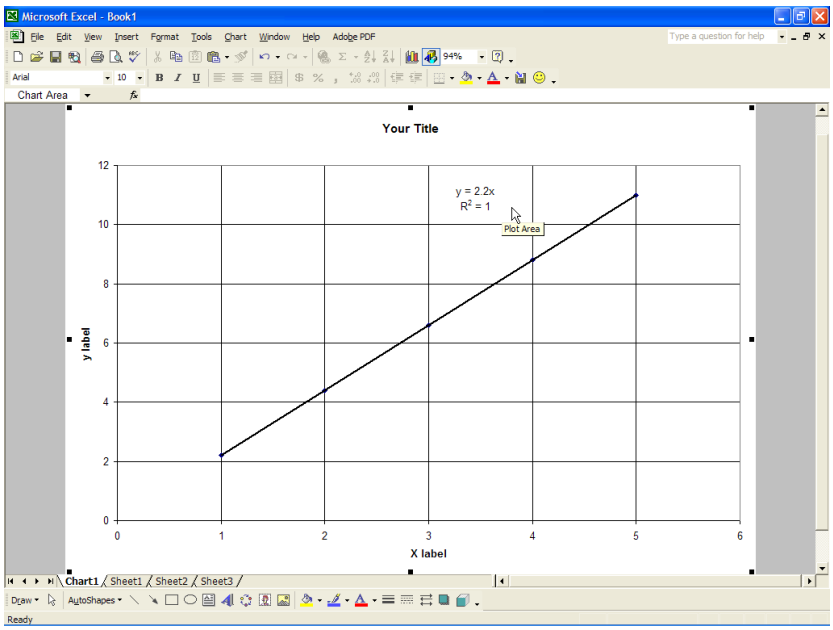



**Figure F-7:** Data with trendline and equation.

# Appendix G. SparkFun LCD

## Connecting the serLCD

An LCD is a display controlled by its own IC. The IC controls which pixels on the display are dark or not. The IC can communicate with other devices (it wouldn't be much use otherwise) but it does not use UART. At the time the LCD was developed, the electronics needed for UART were both expensive and physically much larger. The makers of the LCD, Hitachi, opted for a much simpler control method which requires many more wiring connections. Since UART–enabled chips are now small and cheap, SparkFun developed the serLCD or serial LCD. An extra IC takes UART signals and translates them into signals that the LCD understands. This requires only one wire (besides power for the IC and LCD). The RX or receive pin of the SerLCD should be connected to the TX transmit pin on the PIC18LF2620 using the blue or yellow wire. The power must be connected to +5 V, **connecting it to +3.3 V will not work**. The ground pin can be connects to the breadboard without any buffering or other worries.


**Figure G-1:** SparkFun Serial LCD

The Sparkfun LCD works can operate at different speeds but the default rate is 9600 bits/second or bps. The LCD has two lines of sixteen characters each. When the LCD is connected to power it reinitializes and displays a splash screen for SparkFun. The splash screen can be changed if you like (see the datasheet available on the course Moodle website). Initialization takes about one second and during that time printing to the LCD will have no effect. As a result, you should add a one-second delay to your code after the `configureUART()` call.

## Special Commands

The SerLCD IC interprets the ASCII character decimal 254 (often written as a hexadecimal number 0xFE) as a command character. Anytime you print this character to the SerLCD, the translator IC will interpret the next one character, and only that one character, as a command. For example, the small snippet of code below, in shows how to clear the display and move to the start of the display, turn the underline cursor on, move to the fifth position on the bottom line of the display, and move to the 8th position of the top line.

```
printf("%c%c",254,1);          // Clear and home display
printf("%c%c",254,14);         // Underline on
printf("%c%c",254,128+64+4); // 128 is the command to move to a
                               // specific position and 64 is first
                               // column of the bottom line and 4
```

```
                                // indicates 4 more, i.e. the 5th column.
printf("%c%c",254,128+7);       // top line 8th column
```

The available commands from the SparkFun LCD are given in the table below

| Command | Number code |
|---|---|
| Clear Display | 1 |
| Move cursor right one character | 14 |
| Move cursor left one character | 16 |
| Scroll right (like a billboard) | 28 |
| Scroll left | 24 |
| Turn visual display on | 12 |
| Turn visual display off | 8 |
| Underline cursor on | 14 |
| Underline cursor off | 12 |
| Blinking box cursor on | 13 |
| Blinking box cursor off | 12 |
| Set cursor position | 128 + character position number<br>0 – 15 for line 1<br>64 – 79 for line 2 (or 64 + 0 – 15) |

**Table G-1:** SparkFun LCD command codes

You can also control the splash screen message, toggle it on or on, or change the baud rate of the LCD using the ASCII character decimal 124 (often written as a hexadecimal number 0x7C). Anytime you print this character to the SerLCD, the translator IC will interpret the next one character, and only that one character, as a command. That next character will be a non-printing ASCII character 0 to 31. These non-printing characters are often used as commands by many devices. The SparkFun datasheet available on the course Moodle website uses notation like <*control*>*m*, where *m* may be various symbols, to refer to these non-printing characters. Table B-2 in *Appendix B* lets you translate this notation into the equivalent decimal or hexadecimal number.

To change the splash screen message you could have a program with the lines

```
printf("  Your Message  ");
printf("     Here       ");
printf("%c%c",124,10);
```

You would only use this program once. The message will every time the LCD is rebooted.

Similarly the splash screen can be toggled on or off the next time the LCD booted using the command `printf("%c%c",124,9)`. Again you only do this once.

You can also change the speed at which the LCD can communicate with you PIC MCU. The commands should be sent at the start of the program and take effect the next time the LCD boots. The settings are given below in Table G-2.

| Speed (bits/s) | Command number |
|---|---|
| 2400 | 11 |
| 4800 | 12 |
| 9600 (default) | 13 |
| 14500 | 14 |
| 19200 | 15 |
| 38400 | 16 |
| Reset to 9600 | 18* |

**Table G-2:** LCD baud rate commands

*Note that to send the reset command, the PIC MCU must be set to 9600 bps. If the LCD is not printing correctly, try resetting it with this command.

The special characters used in `printf()` such as `\n` and `\r` do not work with the LCD. Although they are called "nonprintable" characters they actually show as a block or an unfamiliar character on the LCD.

The commands given above are difficult to remember and if you are frequently using the LCD, you would be wise to create a serLCD.h library, a source file and header file, with functions that have easy to understand names. For example, create a function `LCD_ClearDisplay()` that contains the single instruction `printf("%c%c",0xFE,0x01)`. We will also need an `LCD_MoveTo(line, column)` function. Name these files serLCD.c and serLCD.h and place them in your Common directory.

# Appendix H. XC8 and MCC Libraries

The XC8 C Compiler includes version of libraries of functions familiar to users of other compilers. These include math.h, stdio.h, ctype.h, stdlib.h, and string.h. Summaries of the functions in those libraries are given below. More detail is given in *MPLAB_XC8_C_Compiler_User_Guide.pdf - Chapter 8: Utilities*. It also has its own library xc.h that contains functions used to operate the various PICMCU modules. These functions are grouped and described as well.

## math.h

| Function | Description |
|---|---|
| double acos(double f) | Returns arccosine(x) ($\cos^{-1}(x)$) |
| double asin(double f) | Returns arcsine(x) ($\sin^{-1}(x)$) |
| double atan(double x) | Returns arctangent(x) ($\tan^{-1}(x)$) |
| double atan2(double x, double y) | Returns arctangent(x/y) ($\tan^{-1}(x/y)$) |
| double ceil(double f) | Rounds a decimal up to nearest integer. |
| double cos(double f) | Returns cosine(x) (cos(x)) |
| double sin(double f) | Returns sine(x) (sin(x)) |
| double tan(double x) | Returns tangent(x) (tan(x)) |
| double cosh(double f) | Returns hyperbolic cosine(x) (cosh(x)) |
| double sinh(double f) | Returns hyperbolic sine(x) (sinh(x)) |
| double tanh(double x) | Returns hyperbolic tangent(x) (tanh(x)) |
| double eval_poly(double x, const double *d, int n) | Evaluates a polynomial, whose coefficients are contained in the array d, at x, for example:<br>y = x*x*d2 + x*d1 + d0.<br>The order of the polynomial is passed in n. |
| double exp(double f) | Returns the exponential of a number, $e^x$ |
| double fabs(double f) | Returns the absolute value of a number, \|x\| |
| double floor(double f) | Round a decimal down to the nearest integer |
| double fmod(double x, double y) | Returns the remainder of x/y as a floating-point quantity. |
| double log(double f) | Returns the natural logarithm of a number, ln(x) |
| double log10(double f) | Returns the base 10 logarithm of a number, $\log_{10}(x)$ |
| double pow(double f, double p) | Raises a number f to power p, $f^p$. |
| double sqrt(double f) | Returns the square root of a number, $\sqrt{x}$ |
| double trunc(double x) | Rounds to the nearest integer. |

## stdio.h

| Function | Description |
|---|---|
| int printf() | |
| int putchar(int c) | |
| int puts(const char *s) | |
| int getchar(void) | |
| char *gets(char *s) | |

## ctype.h

| Function | Description |
|---|---|
| int isalnum(char c) | c is in 0-9 or a-z or A-Z |
| int isalpha(char c) | c is in A-Z or a-z |
| int isascii(char c) | c is a 7 bit (0 to 128) lower ASCII character |
| int iscntrl(char c) | c is a control character |
| int isdigit(char c) | c is in 0-9 |
| int islower(char c) | c is in a-z |
| int isprint(char c) | c is a printing char (i.e. > ascii 32) |
| int isgraph(char c) | c is a non-space printable character |
| int ispunct(char c) | c is not alphanumeric |
| int isspace(char c) | c is a space, tab or newline |
| int isupper(char c) | c is in A-Z |
| int isxdigit(char c) | c is in 0-9 or a-f or A-F (hexadecimal symbols) |
| char toupper(int c)<br>char tolower(int c)<br>char toascii(int c) | The toupper() function converts its lower case alphabetic argument to upper case, the tolower() routine performs the reverse conversion and the toascii() macro returns a result that is guaranteed in the range 0-0177. The functions toupper() and tolower() return their arguments if it is not an alphabetic character. |

## stlib.h

Note div_t, ldiv_t, udiv_t, and uldiv_t are structures defined in stdlib.h. The members of the structure are quot and rem.

| Function | Description |
|---|---|
| int abs(int j)<br>long labs(long j) | Returns the absolute value of j. |
| double atof(const char *s) | Convert a string to a float |
| int atoi(const char *s) | Convert a string to an 16-bit signed integer |
| long atol(const char *s) | Convert a string to a long integer representation |
| unsigned xtoi (const char * s) | Convert a string to hexadecimal integer representation |
| char *itoa(char *buf, int val, int base) | Convert an integer val to a string buf. Base is either 8, 10, or 16 for octal, decimal, and hex numbers. |
| char *ftoa(float f, int *status) | Converts the float f into a string. |
| char *ltoa(char *buf, long val, int base) | Convert a signed long integer to a string |
| char *utoa(char *buf, unsigned val, int base) | Convert an unsigned long integer val to a string buf. Base is either 8, 10, or 16 for octal, decimal, and hex numbers. |
| double strtod (const char * s, 0) | Converts a string to a float. |
| double strtol (const char * s, 0, int base) | Converts a string to a long. Base is either 8, 10, or 16 for octal, decimal, and hex numbers. |
| int rand(void) | Generate a pseudo-random integer. |
| void srand(unsigned int seed) | Set the starting seed for the pseudo-random generator. |
| div_t div (int number, int denom) | Computes the quotient and remainder of the integer numerator divided by the denominator.<br><br>**Example**<br>*div_t x; // variable structure with two parts*<br>*int quotient, remainder;*<br>*x = div(123, 10);*<br>*quotient = x.quot ; // i.e. 12*<br>*remainder = x.rem,; // i.e. 3* |
| ldiv_t ldiv (long number, long denom) | Computes the quotient and remainder of the long numerator divided by the denominator. See example above. |
| udiv_t udiv (unsigned num, unsigned denom) | Computes the quotient and remainder of the unsigned integer numerator divided by the denominator. See example above. |

| | |
|---|---|
| uldiv_t uldiv (unsigned long num, unsigned long denom) | Computes the quotient and remainder of the unsigned long numerator divided by the denominator. See example above. |

## strings.h

Note, the variable size_t is defined as an integer in string.h.

| Function | Description |
|---|---|
| char *strcat(char *s1, char *s2) | Concatenates or appends string2 to the end of string1. Make sure string1 has enough room! |
| char *strchr(const char *s1, int c)<br>char * strichr (const char * s, int c) | Searches string1 for the first location of the letter 'a'. Returns location. Strichr is case insensitive. |
| char *strcmp(const char *s1, const char *s2)<br>char *stricmp(const char *s1, const char *s2) | Compares the length of two strings<br>< 0 if string1 is less than string2<br>==0 if string 1 is the same as string2<br>> 0 if string1 is greater than string2<br><br>Stricmp is case insensitive. |
| char *strcpy(char *s1, const char *s2) | Copies string2 into string1. Make sure there is enough room. |
| int strcspn(const char *s1, const char *s2) | Calculates the number of consecutive characters from the beginning of string1 that are not contained in string 2<br>string1 string2 result<br>"hello" "aeiou" 1<br>"antelope" "aeiou" 0<br>"antelope" "xyz" 8 |
| int strclen(const char *s) | Counts number of characters in string1 not including NULL |
| strlwr(char *string1) | Convert all uppercase characters to lower case |
| char *strncat(char *string1, char *string2, int n) | Append first n characters of string2 to string1. Make sure string1 has enough room! |
| int strncmp(char *string1, char *string2, int n)<br>int strnicmp(char *string1, char *string2, int n) | Compares the first n character of each string<br>< 0 if string1 is less than string2<br>==0 if string 1 is the same as string2<br>> 0 if string1 is greater than string2<br>strnicmp() is case insensitive |
| char *strncpy(char *string1, char *string2, int n) | Copy the first n characters of string2 into string1. Make sure string1 has enough space! |
| char *strpbrk(char *string1, char *string2) | Search string1 for the first occurrence of the characters in string2. |
| char *strrchr(char *string1,int c)<br>char *strrichr(char *string1,int c) | Finds the last occurrence of the character 'a' in string1, strichr is not case sensitive. |
| size_t strspn(char *string1, char * string2) | Calculates the number of consecutive characters at the beginning of string1 that are contained in the same order in string2 |
| size_t strstr(char *string1, char * string2)<br>size_t stristr(char *string1, char * string2) | Locate the first occurrence of string2 inside string1. Stristr is case insensitive |

## Peripheral Functions and SFRs from MCC

The following functions are generated by the MPLAB Code Configurator (MCC) plug-in for the corresponding peripheral. The functions are described in the header and source files for each.

## Delays

| Function | Description |
|---|---|
| NOP() | Creates a delay of 1 TCY |
| void _DELAY_microseconds(uint16_t number); | Minimum is 40 μs |
| void _DELAY_milliseconds(uint16_t number); | |
| _XTAL_FREQ | FOSC |

## UART

| Function | Description |
|---|---|
| void UART2_Write(char data) | The character to be written to the UART. |
| char UART2_Read(void) | Read a byte (one character) out of the UART receive buffer, including the 9th bit if enabled |
| bool UART2_is_rx_ready(void) | Is date available to be read at RX pin buffer. |
| UART2_DataReady | Identical to UART2_is_rx_ready() |
| bool UART2_is_tx_done(void) | Has transmission finished? |
| bool UART2_is_tx_ready(void); | Is the TX pin buffer busy? Need to wait until it is free to send a message. |

## DIO

Digital pin is RXn where X is port A to E and the specific pin is n = 0 to 7.

| Function | Description |
|---|---|
| IO_RXn_SetHigh(void) | Set digital output high. |
| IO_RXn_SetLow(void) | Set digital output low |
| IO_RXn_Toggle(void) | Switch digital output. If high, set low and vice versa |
| uint8_t IO_RXn_GetValue() | Read idigital input pin status. 1 = high, 0 = low |

## Timers

There are 7 timers, so N = 0 to 7 below.

| Functions and Variables | Description |
|---|---|
| void TMRx_StartTimer(void) | Starts the timer (counter starts incrementing) if not enabled in Easy Setup |
| void TMRx_StopTimer(void) | Stops the timer, counter stops incrementing. |
| uint16_t TMRx_ReadTimer(void) | Reads the timer start value *TMRxH:TMRxL* or *TxTMR* |
| void TMRx_WriteTimer(uint16_t timerVal) | Changes the value of *TMRxH:TMRxL* with TMR0/1/3/5 or *TxTMR* with TMR2/4/6 |

| | |
|---|---|
| void TMRx_Reload(void) | Refreshes the timer with the value of *TMRxH:TMRxL* at initialization. TMR0/1/3/5 only. |
| void TMRx_Period8BitSet(uint8_t periodVal) | Changes the value of *TxPR*. TMR2/4/6 only. |
| TMRx_HasOverflowOccured(void) | Returns value to the flag TMRxIF, which is 1 if the appropriate number of rollovers specified by the postscaler as occurred. |
| TMRxIF | Flag TMRxIF is 1 if the appropriate number of rollovers specified by the postscaler as occurred. Can be read or written. |
| uint16_t timer0ReloadVal16bit | Value of *TMR0H:TMR0L* at initialization. Can be read or written. |
| uint16_t timerxReloadVal | Value of *TMRxH:TMRxL* at initialization. Can be read or written. TMR1/3/5 only. |
| TxCON1bits.CKPS | Prescaler value. Can be read or written. |
| TxCON0bits.OUTPS | Postscaler value. Can be read or written. |

| Timer | Prescalers (N) | Postscalers (M) |
|---|---|---|
| TMR0 | 1, 2, 4, 8, …, 2048, 4096, 8192, 16384, 32768 | 1,2, 3, …, 14, 15, 16 |
| TMR1/3/5 | 1, 2, 4, 8 | None |
| TMR2/4/6 | 1, 2, 4, 8, 16, 32, 64, 128 | 1,2, 3, …, 14, 15, 16 |

Delays: 16-bit timers TMR0/1/3/5 $T = [65536 \times M - startvalue] \times \frac{1}{f} \times N.$

8-bit timers TMR2/4/6 $T = [TxPR \times M] \times \frac{1}{f} \times N$

Intervals ($M$ = 1 & no rollover): $\Delta T = [end - start] \times \frac{1}{f} \times N$

## PWM

Note PWMx, x = 1 to 4, is part of the CCPx peripheral. PWMx, x = 5 to 8, are separate peripherals. PWM period comes from the TMR2/4/6 timers.

| Function | Description |
|---|---|
| Void PWM1_LoadDutyValue(int dutycycle) | dutycycle is an integer from 0 to 1023. Sets the pulse width of the signal. |
| CCPRxH:CCPRxL | Read/Write as CCPRxH × 256 + CCPRxL.<br>CCPx modules only.<br>Same as dutycycle an integer from 0 to 1023 |
| PWMx_INITIALIZE_DUTY_VALUE | Read/Write. Dutycycle is an integer from 0 to 1023.<br>x = 5 to 8 only |
| PWMxDCH:PWMxDCL | Same as above. |

$$T_{PWM} = [PRx + 1] \times N_{prescaler} \times \frac{4}{FOSC}$$

$$T_{dutycycle} = dutycycleValue \times N_{prescaler} \times \frac{1}{FOSC}$$

$$\%DC = \frac{dutycycleValue}{[PRx + 1] \times 4}$$

## Capture

Note x = 1 to 4. Uses TMR1/3/5.

| Functions and SFRs | Description |
|---|---|
| bool CCPx_IsCapturedDataReady() | Has an edge been timed? |
| unsigned int CCPx_CaptureRead() | Read the timer value of captured edge. |
| CCPxIF | Read/Write. Capture flag = 1, when an edge is found. O otherwise |
| CCPxCONbits.MODE | Read/Write. Capture mode.<br>*Value* *Mode*<br>7 Every 16th rising edge<br>6 Every 4th rising edge<br>5 Every rising edge<br>4 Every falling edge |

$$T_{edge2edge} = [T_2 - T_1] \times N \times \frac{1}{f}$$

## SMT

| Function and SFRs | Explanation |
|---|---|
| uint32_t SMT1_GetPeriod(void) | Reads the period, a number between 0 and $2^{24}$ – 1. Flag SMT1IF is set when counter reaches this value. |
| void SMT1_SetPeriod(uint32_t periodVal) | Write the period. Must be a number between 0 and $2^{24}$ – 1. Flag SMT1IF is set when counter reaches this value. |
| void SMT1_ManualTimerReset(void) | Resets the SMT1 counter to zero. |
| uint32_t SMT1_GetTimerValue(void) | Read the current value of the SMT1 counter. |
| void SMT1_DataAcquisitionEnable(void) | Start the counter incrementing. |
| void SMT1_DataAcquisitionDisable(void) | Stop the counter from incrementing. |
| bool SMT1_IsTimerIncrementing(void) | Checks if timer is incrementing. |
| void SMT1_HaltCounter(void) | When counter reaches period, remains at period. Does not roll over. Flag still sets. |
| void SMT1_SingleDataAcquisition(void) | Capture one set of data. Timer stops incrementing after capture completed. |
| void SMT1_RepeatDataAcquisition(void) | Capture data continuously. |
| uint32_t SMT1_GetCapturedPulseWidth() | Reads the CPW buffer. In capture mode, this holds falling edge time. |
| uint32_t SMT1_GetCapturedPeriod() | Reads the CPR buffer. In capture mode, this holds rising edge time. |
| SMT1IF | Timer flag. Equals 1 when the counter reaches period. |
| SMT1PRAIF | PeRiod Acquisition Interrupt Flag. Equals 1 when rising edge detected. |
| SMT1PWAIF | Pulse Width Acquisition Interrupt Flag. Equals 1 when falling edge detected. |
| SMT1CON1bits.MODE | Read/Write SMT1 Modes |

| | |
|---|---|
| | *Value* *Mode* *Pin*<br>0 Timer none<br>1 Gated Timer SMT1SIG<br>2 Period and Duty-Cycle SMT1SIG<br>3 High and low time SMT1SIG<br>6 Time of flight SMT1WIN & SMT1SIG<br>7 Capture SMT1WIN<br>8 Counter SMT1SIG |
| SMT1CON0bits.PS | Read/Write SMT1 Prescaler value<br>*Value* *N*<br>0 1<br>1 2<br>2 4<br>3 8 |
| SMT1CON0bits.EN | Read/Write SMT1 enable status<br>0 = off, 1 - on |

Interval time: $\Delta t = [final\_count - initial\_count] \times N \times \frac{1}{f}$

Delay: $t = period \times N \times \frac{1}{f}$

Capture time: $t = t_{edge} \times N \times \frac{1}{f}$

## ADC

| Function | Description |
|---|---|
| uint16_t ADCC_GetSingleConversion(channel_ANXn) | Return current ADC value, 0 to 1023. X is port, n is pin. |

$$V = \left\{\left(ADCValue + \frac{1}{2}\right) * \frac{[V_{REF+} - V_{REF-}]}{2^n} + V_{REF-}\right\} \pm \frac{1}{2}\frac{[V_{REF+} - V_{REF-}]}{2^n}.$$

## DAC

| DAC1 Function and SFRs | Description |
|---|---|
| void DAC1_SetOutput(uint8_t setting) | Set output pin voltage. Setting is a number from 0 to 31. Voltage determined by Eqn 15-1 |
| uint8_t DAC1_GetOutput(void) | Reads the setting, a number from 0 to 31. |
| DAC1CON0bits.OE1 | Read/Write. DAC1OUT1/RA2 active. 0 = No, 1 = Yes |
| DAC1CON0bits.OE2 | Read/Write. DAC1OUT2/RB7 active. 0 = No, 1 = Yes |
| DAC1CON0bits.PSS | Read/Write. DAC positive reference.<br>*Value* *Selection*<br>3 VDD<br>4 External<br>5 FVR buffer 2 |
| DAC1CON0bits.NSS | Read/Write. DAC negative reference.<br>*Value* *Selection*<br>1 VSS |

| | |
|---|---|
| | 1 External |

$$DAC_{output} = \left\{(V_{REF+} - V_{REF-}) \times \frac{setting}{2^5}\right\} + V_{REF-}$$

## FVR

| FVR SFRs | Description |
|---|---|
| FVRCONbits.RDY | Read only. Is FVR in use? 0 = No, 1 = Yes |
| FVRCONbits.ADFVR | Read/Write. Buffer 1 setting<br>*Value* *Selection*<br>4 FVR off<br>5 1.024 V<br>6 2.048 V<br>7 4.096 V |
| FVRCONbits.CDAFVR | Read/Write. Buffer 1 setting<br>*Value* *Selection*<br>4 FVR off<br>5 1.024 V<br>6 2.048 V<br>7 4.096 V |