

# Simple C for the PIC Microcontroller



**Simon Bramble**

## Copyright Notice

Copyright © 2010 Simon Bramble, [www.simonbramble.co.uk](http://www.simonbramble.co.uk)

All rights reserved

All contents contained within Simple C for the PIC Microcontroller is copyright © 2010 Simon Bramble.

Thank you.

# Contents

<b>Chapter 1: The PIC Architecture</b>	<b>8</b>
The Basics of the PIC architecture Memory Input/Output Pins Interrupts Stack	
<b>Chapter 2: Downloading the C Compiler</b>	<b>10</b>
The HI-Tech editor MPLAB	
<b>Chapter 3: First Steps in Programming</b>	<b>13</b>
Comments #include files Header files Configuration bits The main function	
<b>Chapter 4: Writing Code that Does Stuff</b>	<b>18</b>
for loops while loops do..while loops if statements if..else statements switch statements	
<b>Chapter 5: The First Complete Program</b>	<b>30</b>
Type definitions 0b and 0x notation Disabling the comparator Basic binary manipulation	

<b>Chapter 6: Simulating Code and Programming the PIC</b>	<b>35</b>
Starting a project in MPLAB	
Editing a project using the Hi Tech editor	
Building a project	
Configuring the MPLAB Debugger	
Programming the PIC	
<b>Chapter 7: Timers and Interrupts</b>	<b>49</b>
Configuring the Timer 1 control register	
Configuring the Timer 1 count register	
Configuring the Interrupt register	
<b>Chapter 8: Functions</b>	<b>58</b>
Creating Functions	
Data Types	
Local Variables	
Prototyping	
Variable Scope	
<b>Chapter 9: Working with Binary Numbers and Bits</b>	<b>65</b>
Bits, nibbles, bytes	
Binary numbers	
Hexadecimal numbers	
Logic Functions	
Shifting Bits	
<b>Chapter 10: Data Types</b>	<b>79</b>
Unsigned char	
Unsigned int	
Converting data types	

Chapter 11: Arrays	82
Single Dimensional arrays	
Multidimensional arrays	
Storing text	
Driving LCDs	
Chapter 12: Other Useful Snippets	100
The enum keyword	
The static keyword	
The volatile keyword	
Appendix A: Variable Names	102
Appendix B: C Operators and their Shortforms	103
Appendix C: The <code>#define</code> Directive	106
Appendix D: Suggested LED Schematics	108
Appendix E: Suggested LCD Schematic	111
Appendix F: Preprocessor Directives	112
Appendix G: ASCII Codes	113

# Introduction

Microprocessors are becoming more complex year on year as are the tasks that are required of them. While it is acceptable to obtain a grasp of the workings of a microprocessor using assembly language, if more complex tasks are to be performed, a simpler language is needed. In addition, an engineer can cut his teeth on one processor, become proficient in its assembly language but then be a complete novice if he needs to upgrade to a processor from a different manufacturer that uses a completely different set of instructions. A universal language is needed that is not processor dependent that enables an engineer to write software regardless of the platform and regardless of the complexity of the task in hand.

This is where the C programming language comes to the rescue. C was designed to be simple, yet flexible enough to allow engineers to perform complex tasks with little knowledge. Moreover, because it has been widely adopted by the engineering community, many microprocessor suppliers have employed compiler companies to design compilers based on their particular architecture. Therefore, once an engineer becomes proficient in the C programming language, he can switch from one microprocessor to another with little effort and use simple and complex processors with similar ease. The C compiler handles the problem of translating high level C into lower level assembly code (and thence the native machine code).

The problem with C compilers is that they are beyond the budget of most engineers trying to learn C, so assembly language still reigns supreme in the hobby market. C is kept to those engineers employed in companies not afraid to spend thousands of dollars on compilers. This is a pity, because although C occupies more memory space, so much more can be achieved with C over assembly language and indeed an engineer with knowledge of the basics of C can explore many more applications and processors expanding his electronics knowledge as he goes.

Another reason why some hobby engineers do not adopt the C programming language is that it can look intimidating. Ironically many proficient C programmers think the same of assembly language and baulk at having to write hundreds of lines of assembly code to perform a simple task. However, C does not have to be complex and many complicated tasks can be performed with only the slightest knowledge of C. In fact, this is often the best way of learning any language (computer or spoken): start with the simple stuff, build your confidence with this and slowly learn the complicated aspects as your confidence grows.

Finally, many C programming books are aimed at writing PC based software, so many of the programming examples include functions that scan a keyboard and print the results. In the embedded world (software designed to run on a microprocessor inside a piece of equipment that does not necessarily have a keyboard, mouse, printer and screen) the print and scan functions are completely redundant.

## Introduction

This book has 3 aims:

Firstly to present to the reader a way of obtaining a legal, free of charge C compiler and show the reader how to install the compiler and write a simple C program for a PIC microcontroller.

Secondly to demonstrate that complex C programs can indeed be written using only the simplest of C language commands. This will enable the reader to build confidence with the C programming language in order to move on to other books that teach the more complex aspects of C.

Thirdly to provide the reader with code snippets that have been tried and tested on the PIC microcontroller and actually work. As with learning a conventional language (like English) you learn by copying then doing and these code examples will back up the theory so the reader can become proficient in writing simple code in C for embedded applications. This book has been written to get the reader exposed to code as quickly as possible so he can start writing his own programs (however rudimentary) almost immediately. This has been done at the expense of a more traditional logical flow of chapters. Rather than wade through chapters of theory, this book dives into code almost immediately with the finer details of the code explained in later chapters.

In writing this book, the author has considered anyone completely new to microprocessors and included a brief chapter on how microprocessors work in general. This will help the reader understand the mechanics of a microprocessor, but it is not essential to read this chapter first. It is perfectly acceptable to dive straight into the coding chapters and refer back to the basics if needed.

All of the code in this book has been based on the PIC16F627A microprocessor. It is worth downloading the latest copy of this datasheet from

[www.microchip.com](http://www.microchip.com)

to see what components inside the PIC16F627A are being referred to.

You will also need a PIC16F627A I/P chip. These can either be purchased from any good electronics component supplier or ordered as a sample from the Microchip website. It is worth ordering 2 just in case things go wrong..

Finally, since this book is available in electronic format, it can be changed and updated very easily to suit the reader. If you have any comments on improvements that can be made to this book, please email us at:

[enquiries@simonbramble.co.uk](mailto:enquiries@simonbramble.co.uk)

# Chapter 1

# The PIC Architecture

## The Basics of the PIC architecture

Although it is not essential to know every last detail of how the PIC microcontroller works, it helps to have an understanding of memory, ports, interrupts, and peripherals and how they all interact.

### *Memory*

The PIC has 2 types of memory: ROM (Read Only Memory) and RAM (Random Access Memory).

ROM is memory that holds your program. Once it is programmed, it cannot be modified by program execution. It can be erased and reprogrammed (if you want to change your software and reprogram the microprocessor), but during execution of your program, the ROM remains unchanged. There are several types of ROM, but the type most PICs use is Flash ROM. This daft name comes from the fact that its predecessor took ages to erase and program, whereas Flash memory can be erased and programmed quickly (in a Flash). If you remove the power to the microprocessor, the ROM remains programmed, so that your program can restart when the power is reapplied.

RAM memory is where all the variables of a program are stored. If you are counting the number of times a particular part of code is executed, the count value is stored in RAM. If you remove the power to the processor, all RAM data is lost.

### *Input/Output Pins*

The most fundamental parts of any microprocessor are the port pins. These interface your program to the outside world, so the first step in learning any computer language or microprocessor is to get the port pins moving. The simplest application for any microprocessor is to flash an LED and this will be our first code example. This simple application proves that we have been successful in several tasks: we have found a program that enables us to enter code, program some registers to toggle the port pins, compile it without errors, download it to the target microprocessor, power up the processor and run the program.

Simplistically, programming a microprocessor is nothing more than programming its internal registers to do stuff. If you can program the microprocessor registers to set port pins high and low, you can do pretty much anything you want within the capabilities of that particular processor. Being able to toggle the port pins is a very big step forward in mastering the complete microprocessor.



### *Interrupts*

To keep things simple, it is convenient to imagine that the program you write starts at address 0000 and steps through memory addresses 0001, 0002... and so on. Many microprocessors have interrupts included in their architecture. These are incredibly handy functions as they allow the program to flow normally until some outside event occurs. When the event happens, the microprocessor jumps to a known location where the programmer has written software (known as an Interrupt Service Routine) to handle the interrupt. The interrupt is dealt with and normal program execution is resumed. Fortunately for the programmer, the C compiler handles all the addressing and jumps to and from the interrupt service routine. All the programmer has to do is worry about writing the code to handle the interrupt, but this will be covered later. A good example of this is if a program needs to perform a function every 2 seconds. The programmer can set up a timer to count a 2 second period then generate an interrupt. Thus the microprocessor can perform its other tasks (for example measuring a temperature and displaying the results on an LCD) and every 2 seconds saving the temperature to external memory or flashing an LED, or sounding a buzzer.

### *Stack*

Software is normally written as a series of functions. So for a temperature logger, you will write a function that reads in the analogue voltage from the temperature sensor, there will be another function to convert this to a series of numbers to display on the LCD, there will be one to drive the actual display and finally, possibly a function to time how often the reading takes place (once per second for example). All of these functions are generally 'called' from the main program. The program starts by stepping through its instructions (at address 0000, 0001, 0002...). When a function is called, the current address of the program is stored in an area of memory called the stack and the program jumps to the address where the function is located. When the function has been executed, the microprocessor looks to the stack to see what address to return to. One function call takes up one place on the stack. If a call is made to a function and then inside that function *another* call is made to another function, 2 places are taken up on the stack and so on... A 2 level deep stack allows a 'call within a call' to be made. So a stack is a temporary storage space for addresses to be stored by the processor during the execution of the program.

Fortunately much of the addressing issues outlined above are handled by the C compiler, so the programmer does not have the worry of locating code in ROM/RAM or at any specific memory address. Nor does he have to worry about his code jumping to the correct address during an interrupt or handling the stack during function calls.

# Chapter 2

# Downloading the C

# Compiler

In this chapter we will download free of charge tools to allow you to write C code, compile it, debug it and program a PIC microcontroller with it.

Until recently, Hi Tech provided a very good compiler and editor that could be used to write and debug code. Unfortunately the simulator (in our experience) was not as good as the one offered by Microchip. This meant that code was written using the Hi Tech editor and simulated in Microchip's MPLAB environment. There was also a program called Universal Toolsuite to link the two programs to ensure that a modified file in the Hi Tech editor appeared as a modified file in MPLAB.

All that has changed since Microchip bought Hi Tech. Now everything is done within the Microchip environment and the latest version of MPLAB includes the Hi Tech C compiler and can be downloaded free of charge from the Microchip website. However, we still feel that it is easier to write code in the editor from Hi Tech, then compile and simulate it in MPLAB. Since you cannot obtain the editor from Hi Tech anymore, we have provided it on the SimonBramble Website:

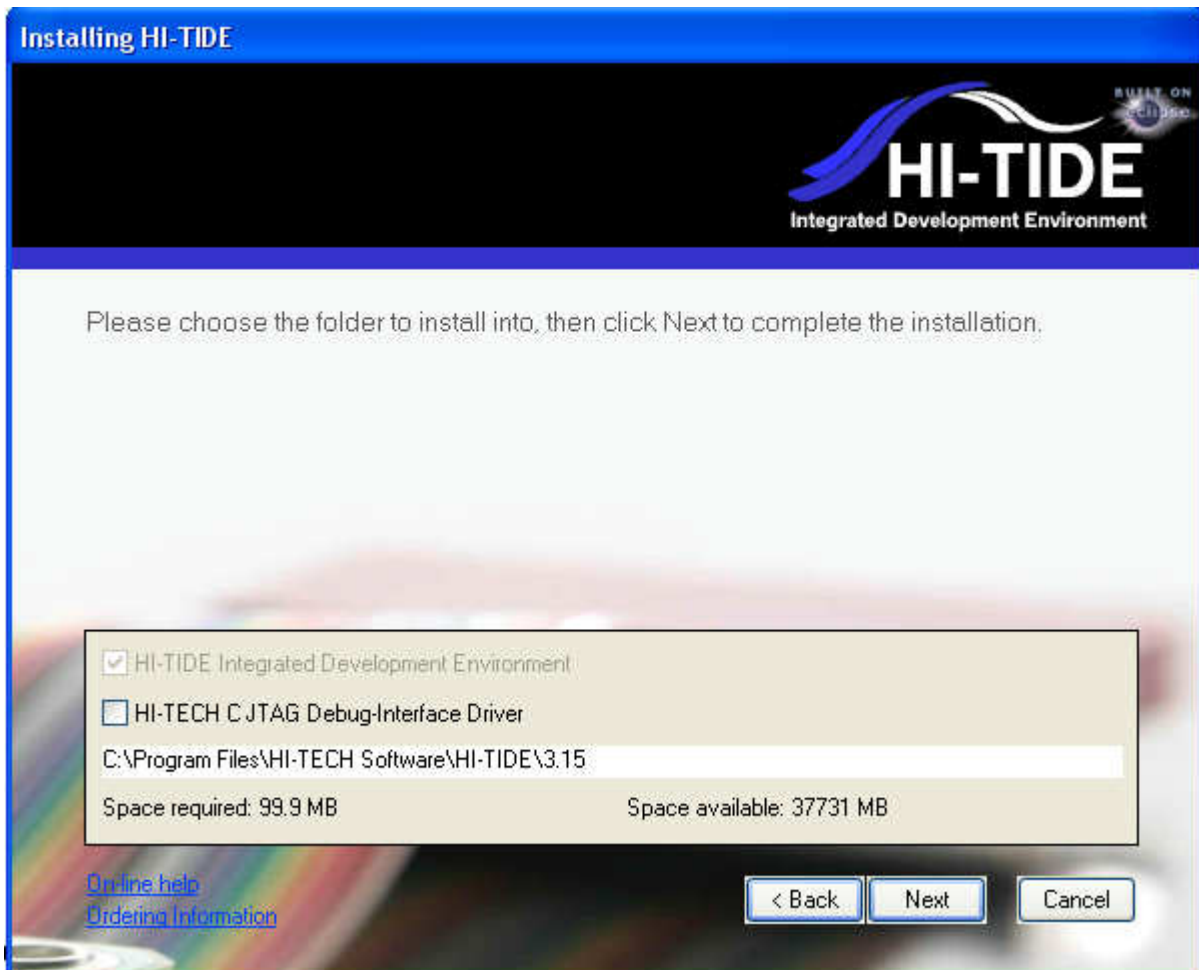
[www.SimonBramble.co.uk](http://www.SimonBramble.co.uk)

We suggest you write the code in the *editor* provided by Hi Tech and use MPLAB to compile and simulate your code as well as download your code to the target microprocessor.

To install Hi Tech's editor (HI TIDE Integrated Development Environment), run the executable file, hi-tide\_v3.15.exe downloaded from our website and follow the instructions.

When prompted to install HI TECH C JTAG Debug Interface Driver, unselect the checkbox as this is not needed.

## Downloading the C Compiler



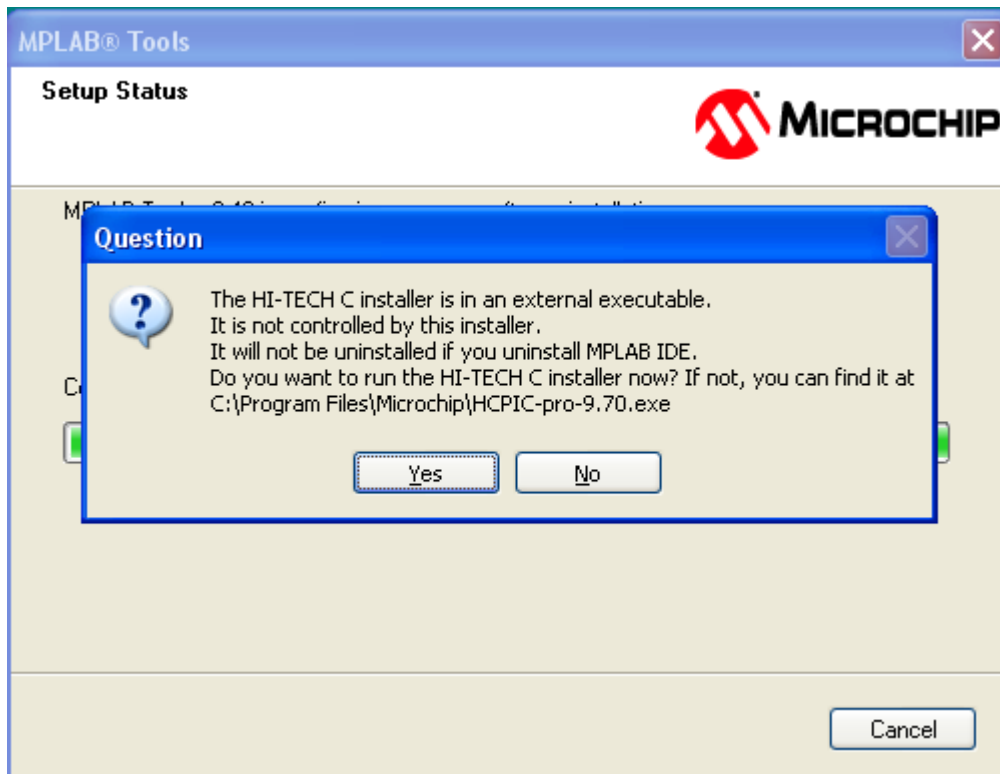
The Universal Toolsuite will be installed when you install MPLAB, thus any changes made in your editor will automatically be updated in MPLAB.

To download MPLAB, go to Microchip's website at:

[www.microchip.com](http://www.microchip.com)

Their homepage is constantly changing, but look for 'MPLAB IDE'. This is Microchip's Integrated Development Environment. Download MPLAB 'Full Released ZIP version' and save it in a folder of your choice.

To install MPLAB, unzip the files and follow the instructions. When you get greeted with:



Select 'Yes' regardless of whether you have already installed the Hi Tech C compiler as this will install any additional components that MPLAB might need. How to use MPLAB will be outlined in a future chapter.

# Chapter 3

# First Steps in

# Programming

Let's get down to learning how to program. The good thing about C is that it is a 'high level' language. Basically, low level languages have instructions close to machine code (the 1's and 0's that the microcontroller works on). Higher level languages are nearer to English. The C compiler translates high level code into 1's and 0's, so the programmer does not have to worry too much how the microprocessor works. With C being a high level language, much of the programming technique becomes instinctive after a while as C is very close to our native English.

Let's start with a skeletal program on which we will build our program. Don't worry too much about not understanding the following few pages immediately. The following template can be copied across from one program to the next and understood over time.

A good template to start with is the following:

```
/*-----*/
/* Program Name:  LED Flasher                      */
/*-----*/

#include <htc.h>

__CONFIG(PROTECT & CPD & LVPDIS & BOREN &
          MCLRDIS & PWRTEN & WDTDIS & INTIO);

void main(void)
{

}
```

Starting at the top, you will notice a comment (in green). Comments are completely ignored by the compiler. Comments at the top of the code normally include the name of the file, the author, the date the code was written, code revision, type of processor used, clock frequency etc. You can put as many comments throughout your code as you feel necessary and some programmers say the

more comments the better. It helps explain your code to anyone reading it and can also be useful if you need to go back to your code at a later date and cannot work out what your code means (yes this does happen).

In C a comment is defined by `/*` at the start of the comment and `*/` at the end of the comment. Anything between these lines is classed as the comment, *including carriage returns*. Therefore the above comment can span multiple lines with just one `/*` at the start and a `*/` at the end as shown below:

```
/*-----  
Program Name:    LED Flasher  
-----*/
```

It looks neater if the comment is terminated at the end of each line with a `*/` and the next line started with a new `/*`, but this is personal preference.

You might also see comments written as follows:

```
//-----  
// Program Name:  LED Flasher  
// -----
```

Here a `//` is used at the start of every line. However, here the comment is terminated at the carriage return, so a new `//` is needed at the start of every new comment line. Strictly this does not conform to the C standard and some purists will say it is illegal. However, most compilers accept it.

Personally, to keep your code conforming to the standard, only the `/*...*/` format should be used.

The next line of our skeletal program is:

```
#include <htc.h>
```

Now, we mentioned that C is a high level language and one of the benefits of using a high level language is its readability. Some programmers go to extraordinary lengths to make their code readable, often at the cost of code size. The good thing about C is that the programmer can write several functions, all in separate files and tell the C compiler to compile them all together as though they were one big file. This has huge benefits in the readability of the code and a large program will be divided into lots of smaller functions in separate files often only a few lines long. Moreover, if a programmer writes a particularly useful function, this can be copied over into another completely different program and used again, thus saving having to rewrite code from scratch. This benefit of being able to hive off code into other files is a major benefit of C.

## First Steps in Programming

Header files (defined by the '.h' on the end of the file name) are used to define variables and addresses that are used across several different files. Instead of defining these variables in each file, we put them all into a header file and just 'include' the header file (using the statement `#include`) at the top of each file. The compiler then looks in the header file for the variable definitions. The overall effect is that the code looks very much neater with all the variable definitions out of the way in a separate file.

As an example, your code might write a value to PORTA. PORTA is located at address 0005hex. Rather than your code having statements like:

```
setport 0005hex
```

It is far more readable if your code reads as follows:

```
setport PORTA
```

If you define PORTA as being equal to an address 0005hex in the header file, then every time the compiler reaches a statement including 'PORTA' it looks to the header file to see what the definition of PORTA actually is. It then replaces all instances of PORTA with 0005 and compiles the code. Therefore when reading through your code, you know exactly what your code is doing (operating on Port A) instead of having to decipher some strange register address. The file above ([htc.h](#)), contains all the definitions of the port pins, register bits, register names etc are defined which make your code easier to read.

(Technically, [htc.h](#) calls up another file that contains all the definitions, but to keep things simple, it can be assumed that [htc.h](#) contains all the definitions. For the more curious reader, the [htc.h](#) file is located in the `c:\program files\hi-tech software\picc\9.70\include` directory.

Inside the [htc.h](#) file, there is a line

```
#include <pic.h>
```

telling the compiler to look in the [pic.h](#) file. Inside the [pic.h](#) file, there is a line

```
#include <pic16f62xa.h>
```

telling the compiler to look in the `pic16f62xa.h` file. Inside this file is where all the 16F627A registers are defined, including a line that tells the compiler to replace all 'PORTA comments with the more mysterious address of Port A, 0005.)

So here we are making use of C's ability to combine multiple files to get rid of a lot of variable definitions that would otherwise clutter up our program and make it difficult to read.

The way to include the header file in your main code is to use the `#include` statement and if used, should always be placed at the very top of the code. This is known as a *preprocessor directive* and it tells the compiler to include certain definitions specified in another file. Preprocessor directives start with the `#` symbol and `#include` is the only one you need for now. All the Hi Tech preprocessor directives are outlined in Appendix E.

Another benefit of a header file is that you can put any port definitions and program constants into the header file, then if they need to be changed at a later date, you only need to change one entry in a single file instead of having to open up each file to see if it uses that constant. Port pins are often quite randomly chosen in some PIC programs (there are no difference between Port A and Port B on the PIC16F627A), but if it is easier to lay out our PCB such that the LED is driven from Port B bit 0 instead of Port A bit 0, changing a single header file is easier than changing each reference to that port within all files.

So a header file is a file that is included in your main code and makes your code easier to read.

The next line sets up the configuration bits of the microprocessor.

```
__CONFIG(PROTECT & CPD & LVPDIS & BOREN &  
         MCLRDIS & PWRTEN & WDTDIS & INTIO);
```

The above statement is specific to the PIC compiler we are going to use. Don't worry about its syntax too much as this line can be copied over from program to program and only modified slightly if a new processor is used. The above statement tells the compiler to program the processor with the following attributes:

- Code protect: enabled
- Protect Data memory: enabled
- Disable Low voltage programming: enabled
- Brown out detect: enabled
- Master clear pin: disabled
- Power Up Timer: enabled
- Watchdog timer: disabled
- Oscillator is internal



## First Steps in Programming

How these configuration bits affect the PIC microcontroller can be found in the datasheet.

Again, these words (`PROTECT`, `CPD`, `LVPDIS` etc) are defined in the header file `htc.h`.

And finally next is the all important `main` function:

```
void main(void)
{

}
```

Every program written in C has to have (only one) `main` function. This tells the compiler where the main program starts and takes the form of any other function in C which will be explained later.

The first word in any function defines the *output* of that function – what result the function produces, if any. The word `void` tells us that this particular function gives out nothing. Next is the name of the function and our main function has to be called `main`.

Next (in brackets) is a list of variables that the function takes as *inputs*. In this case, our `main` function takes in no variables and we define this by using the word (`void`).

Any function then contains one or more lines of code that define what the function does. Most of the time a function will have more than one line of software and these are grouped together in brackets thus: `{}`. If there is just one line in the function then curly brackets are not needed. It is, however, good to get into the habit of always putting brackets around the function code even if it is one line as this makes code readability easier.

In summary, this chapter has defined our skeletal program. This can be used as a template for all future programs you write with the only thing that changes being the code we are going to insert between the curly brackets in our `main` function.

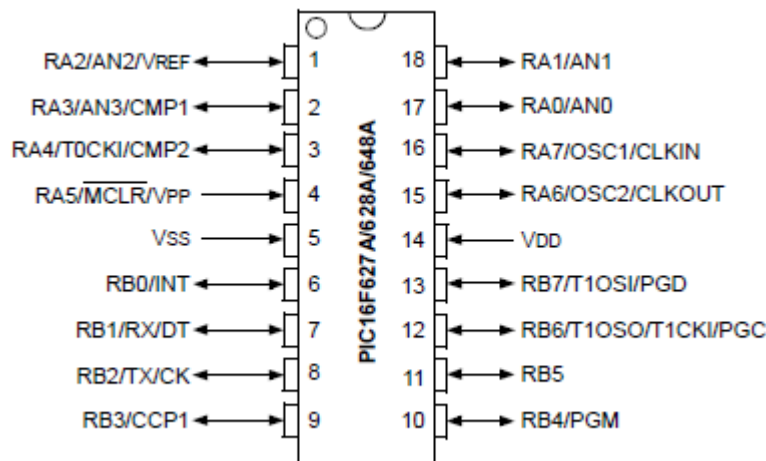
# Chapter 4

# Writing Code that

# Does Stuff

A microprocessor (put simply) is nothing more than a series of registers that the user programs to do stuff. The simplest task a microprocessor can do is switch on and off an LED on a port pin and once we have proved that we can use the internal registers to flash an LED on and off, programming the other registers inside the microcontroller is simple. The C programming language makes it incredibly simple to change the state of a port pin on the PIC microcontroller and this can be achieved using one line of code.

Firstly, the PIC16F627A has two 8 way (or 8 bit) port pins, RA0 to RA7 and RB0 to RB7 as shown below. Please note that the port pins start with the suffix '0', so referring to Port A bit 1 actually refers to the *second* I/O pin of Port A. These port pins are shared with other functions, but for the moment, we will ignore these.



A binary number can be written to Port B and this will appear as a voltage (either 5V or 0V) on the pins of the microprocessor. Binary numbers are explained more in the chapter Working with Binary Numbers and Bits. The line

## Writing Code that Does Stuff

```
PORTB = 0b11110000;
```

sets Port B bits 7, 6, 5, 4 to logic '1' (5V) and Port B bits 3, 2, 1, 0 to logic '0' (0V). It is that simple. The '0b' is simply a way of telling our compiler that the number that follows is a binary number. We will now demonstrate all of the important functions in C using the LED flashing principle.

Most computer programs run from start to finish executing statements as they go. There are a few program statements that make decisions along the way and change the flow of the program depending on the outcome of the decision.

You can write quite sophisticated programs with only 3 program statements. These are:

```
for (...)
{
    /* put your instructions here */
}

while(...)
{
    /* put your instructions here */
}

if (...);
{
    /* put your instructions here */
}
```

### for Loops:

If an instruction or group of instructions need to be executed a fixed number of times, the **for** program statement is ideal. An example is shown below.

```
for (n=0; n<10; n++)
{
    /* put your instruction to be repeated here */
}
```

The first line is the *evaluation* and it assigns the value 0 to a counter, n. (any integer variable can take the place of n and this will be explained later). The instructions within the curly brackets are then executed, n is incremented (using the statement n++), the evaluation is performed again and the instructions are executed while n<10. The 'n++' is the same as writing

```
n = n + 1
```

in other words, increment n by 1, so the instruction is executed 10 times.

Equally, the following statement does the same thing:

```
for (n=1; n<11; n++)
{
    /* put your instruction to be repeated here */
}
```

Likewise:

```
for (n=10; n>0; n--)
{
    /* put your instruction to be repeated here */
}
```

The above lines assign the value 10 to a counter, n, execute the instructions within the curly brackets, *decrement* n and continue to execute the instruction while n>0. The 'n--' is the same as writing

```
n = n - 1
```

in other words, decrement n by 1, so the instruction is executed 10 times. Note also the change of the < sign to > as we are executing the statements while n>0.

All 3 statements above perform exactly the same function, although the first one is probably more instinctive in how it operates.

An actual example of the **for** statement is shown below.

```
for (n=0; n<10; n++)
{
    PORTB = 0b11111111;
    PORTB = 0b00000000;
}
```

This `for` statement assigns the value of 0 to a counter, `n`. The statement then executes the lines of code inside the curly brackets. These lines assign a value of all 1's to Port B, then assign a value of all 0's to Port B. If LEDs were connected to PortB, they would flash on and off 10 times. Again, note the use of the '0b' notation before our 8 binary bits telling the compiler that the number we are dealing with is a binary number with each binary digit (bit) representing a port pin.

Note the layout of the code. The code inside the curly brackets has been indented. The compiler ignores this indentation, but it MUST be included to make the code more readable. The curly brackets containing the code relevant to the `for` statement are directly below that `for` statement. Nothing else exists on this line. The code inside these curly brackets starts on the line below and is indented. Thus it can easily be seen what code is applicable to the `for` statement. At the end of the `for` statement, the closing curly bracket is in line with the opening curly bracket – i.e. not indented. It is not so important with this example, but if you get a `for` statement inside an `if` statement inside a `while` statement, things can get a bit unreadable. In cases like this, indentation plays an essential part in the readability of the code.

It is also worth noting that normal lines of code have to be terminated in a semicolon and program statements such as `for`, `while` and `if` do not have a semicolon at the end.

`for` loops can be 'nested' and this is particularly helpful in creating a delay routine. The code above will turn PortB on and off, but at such a fast rate, that the eye will not be able to see the LED flashing, assuming the LED can even keep up with the speed! To give us some idea of the speed, the default clock speed of the PIC microcontroller is 4MHz, or one clock cycle every 0.25us. It takes four clock cycles to execute most assembly language instructions so an instruction cycle executes every 1us. Each C instruction breaks down to several assembly language instructions so we can immediately see that our code above is going to flash our LED on and off every few microseconds – far faster than the eye can see.

Assuming we need the LED to flash once per second (on for half a second and off for half a second), we need to turn on the LED, have a `for` loop to create a delay of 0.5 seconds, turn off the LED then have another delay of 0.5 seconds. We start with the code below:

```
for (n=0; n<255; n++)
{
    PORTB = 0b11111111;
}
```

Our value of `n` can only count up to 255 (this will be explained later). Putting this code into MPLAB and simulating it shows that it takes 3.34ms for `n` to count from 0 to 255. Putting another `for` loop around the above loop will ensure that our 3.34ms loop occurs multiple times. We calculate the loops needed as follows:

$$\text{Loops} = \frac{0.5 \text{ seconds}}{3.4 \text{ms}} = 150$$

The code below will achieve this:

```
for(x=0; x<150; x++)
{
    for (n=0; n<255; n++)
    {
        PORTB = 0b11111111;
    }
}
```

Starting with the ‘inner’ loop, we are writing a binary value of 1111 1111 to Port B 255 times. We then repeat this complete sequence 150 times, so in effect we are writing to Port B 38250 times. We have nested the loop of 255 counts inside another loop of 150 counts.

Simulating this code in MPLAB shows that the LED is ON for 499.5ms.

We now need to turn the LED OFF, so can repeat the same exercise. The loop below takes 4.08ms (simulated in MPLAB) to count from 0 to 255.

```
for (n=0; n<255; n++)
{
    PORTB = 0b00000000;
}
```

123 cycles of this loop will give us our OFF time of 500ms, thus:

```
for(x=0; x<123; x++)
{
    for (n=0; n<255; n++)
    {
        PORTB = 0b00000000;
    }
}
```

## Writing Code that Does Stuff

Note the indentation. All code applicable to the first **for** statement is indented once. All code applicable to the next **for** statement is indented twice. With this indentation method, it is easy to see code inside each **for** statement.

Suppose we want to flash our LED 5 times. Putting in another **for** statement allows us to do this:

```
/* flash LED 5 times */
for(i=0; i<5; i++)
{
    /* turn LED ON for 0.5s */
    for(x=0; x<150; x++)
    {
        for(n=0; n<255; n++)
        {
            PORTB = 0b11111111;
        }
    }
    /* turn LED OFF for 0.5s */
    for(x=0; x<123; x++)
    {
        for(n=0; n<255; n++)
        {
            PORTB = 0b00000000;
        }
    }
}
```

Now you can see the importance of indentation! The code for each **for** loop is contained within its own set of curly brackets and each closing bracket is directly below its associated opening bracket. It is also worth pointing out that each loop uses a different variable counter (*i*, *x* and *n*). If your program starts doing strange things, check that you have not used the same variable twice. Note also the comments, telling the reader exactly what the next block of code does.

There are easier ways of generating delay routines and these will be outlined later, but for now, this is a fine example of creating a delay routine with a nested **for** loop.

### while Loops

The next statement in our armoury is the **while** statement. As it would suggest, this executes a group of instructions *while* a condition is true, regardless of how many times that might be. Here is an example.

```
n = 0;
while (n < 10)
{
    PORTB = 0b11111111;
    PORTB = 0b00000000;
    n++;
}
```

Again we notice that each line of code within the brackets has a semicolon at the end whereas the program statements do not.

Analysing this function, we start with the evaluation `while(n < 10)`. If the `while` statement is true, the statements inside the curly brackets are executed. The evaluation is then reassessed and the statements are executed again as long as the evaluation is true. It may seem obvious, but there must be a way of modifying the variable, `n`, otherwise the loop will continue forever. This is unlike the `for` loop that has the increment or decrement as part of the `for` statement at the top of the loop.

Now, sometimes a group of statements needs to be executed *before* the evaluation takes place. A simple modification of the `while` statement allows us to do this, as follows:

```
n = 0;
do
{
    PORTB = 0b11111111;
    PORTB = 0b00000000;
    n++;
}while (n < 10);
```

As always, each line has a semicolon at the end, except for the evaluation. There is also a semicolon after the `while` statement. Now, some might argue that the code is more readable if the `while` statement is placed below the last curly bracket. However, putting it on the same line groups it together with the previous `do` statement and its associated curly brackets. If it were placed on the line below, the reader might think that it was a new `while` statement and wonder why it had a semicolon at the end.

Another way of looking at the `while` statement is as a True/False evaluation. In C, the compiler assigns the evaluation a value of 1 if the evaluation is True and 0 if False. Sometimes our program might want to execute an infinite loop (loop forever) and we can use the `while` statement with a forced value of 1 (always true) to do just that with the statement below:



## Writing Code that Does Stuff

```
while(1)
{
    /* put your instructions here */
}
```

Any code inserted inside the curly brackets will be executed forever.

To be more precise, the **while** statement will execute if the evaluation is *non-zero*. In the case of a comparison, if the answer is true a value of 1 is assigned to the evaluation and if the answer is false, a value of 0 is assigned to the evaluation. However, if the brackets were filled with any non zero value then the **while** statement would still execute, so

```
while(10)
{
    /* put your instructions here */
}
```

has the same effect as

```
while(1)
{
    /* put your instructions here */
}
```

### if Statements

Unlike the **for** and **while** statements, the **if** statement is a single conditional test, so evaluates the **if** statement, then executes the instructions inside the curly brackets once on the condition that the **if** statement is true. Of course, the statement can be nested inside another statement that will force it to loop as the following code illustrates

```
n = 1;
while(1)
{
    n++;
    if(n==10)
    {
        n = 0;
    }
}
```

The above code illustrates a number of issues we have discussed so far. Firstly, the variable, *n*, is assigned the value 1. The next line is a **while** statement and only executes if the **while** condition is true. In this case, because the condition is always true (we have assigned it a permanent value of 1), the loop will continue to execute permanently – it is an infinite loop.

The variable *n* is increment by one to the value 2. Then the **if** statement is reached. The code within the **if** curly brackets is only executed if *n* is equal to 10. In the first parse through this code, *n* is equal to 2, so the code within the **if** statement is not executed. The code then reaches the end of the **if** curly brackets, falls out of the **if** statement, then reaches the end of the **while** statement, so loops back to the **while**(1) evaluation and the process starts again. *n* is then increment to 3 and so on... It can be seen that as soon as *n* reaches the value 10, the **if** statement becomes true, the code within the **if** statement is executed and *n* is given the value of 0.

Note the difference in the use of the ‘equals’ signs. A single = sign is an *assignment* – it assigns a number to a variable. A double equals sign (==) is a comparison – it compares one variable (or value) with another variable. It can be seen that a == is needed in any conditional statements, like **if**, as the variable inside the brackets (*n*) is being compared, not assigned. Some compilers (including the Hi Tech compiler) will only throw up a warning if you use an **if** statement with a single equals sign, but will still compile the code. Beware: A simple mistake like this can take hours of frustration working out why your code is misbehaving. Refer to Appendix B for more examples of the difference between == and =.

Suppose we want to execute a group of statements if one condition is true, but execute another set of statements if another condition is true. This can be achieved by expanding the **if** statement with an **else** statement.

```
n = 1;
while(1)
{
    if(n==10)
    {
        n = 0;
    }
    else
    {
        n++;
    }
}
```

The above code is the same as the last example, but this time increments *n* in all cases other than when *n* is equal to 10 where it sets *n* to zero. As with the **while** statement, the **if** statement can evaluate to any non-zero value in order to execute and does not have to be ‘1’ or ‘0’. This is an important point to note when it comes to bit manipulation and will be discussed later.

## Writing Code that Does Stuff

The **if** statement is good for executing a set of statements given the outcome of a single evaluation and the **if else** statement enables us to choose two different sets of instructions determined by a single evaluation. Now let's suppose we have multiple evaluations to make. This could be achieved by cascading a series of **if** statements, but the code might start to look messy. The following code uses a **for** loop to increment a variable, *n*, from 0 to 4. During each increment, it evaluates *n* and assigns another variable *x* to a multiple of *n*.

```
for (n=0; n<5; n++)
{
    if(n == 0)
    {
        x = 2*n;
    }
    if(n == 1)
    {
        x = 3*n;
    }
    if(n == 2)
    {
        x = 4*n;
    }
    if(n == 3)
    {
        x = 5*n;
    }
    if(n == 4)
    {
        x = 6*n;
    }
}
```

This is perfectly valid code, but there is a neater way of achieving the same goal. The **switch** statement provides a useful alternative to multiple **if** statements.

```
for (n=0; n<5; n++)
{
    switch(n)
    {
        case 0: x = 2*n; break;
        case 1: x = 3*n; break;
        case 2: x = 4*n; break;
        case 3: x = 5*n; break;
        case 4: x = 6*n; break;
        default: x = x; break;
    }
}
```

Here the variable,  $n$ , inside the `switch` evaluation has to evaluate to type `char`, `int` or `long` or any `unsigned` variants of these. Once  $n$  is evaluated, the appropriate `case` statement is executed and the code continues to execute until a `break` statement is reached, after which the `switch` statement is exited.

For example, in the above program, when  $n$  becomes equal to 3, the code jumps to the line

```
case 3: x = 5*n; break;
```

$x$  is assigned the value  $5*n$ , the `break` statement is reached and the `switch` statement is exited.

If a `break` statement is not included at the end of a line, the code will flow down to the next `case` statement and that line will be executed too, thus in the following code

```
for (n=0; n<5; n++)
{
    switch(n)
    {
        case 0: x = 2*n; break;
        case 1: x = 3*n; break;
        case 2: x = 4*n; break;
        case 3: x = 5*n;
        case 4: x = 6*n; break;
        default: x = x; break;
    }
}
```

if  $n$  evaluates to 3 the following line is executed

```
case 3: x = 5*n;
```

and since there is no `break` statement the code flows down to the line

```
case 4: x = 6*n; break;
```

The net effect of this is that  $x$  is assigned the value  $5*n$ , then assigned the value of  $6*n$  as the `case 4:` condition is executed. We can use this to our advantage in situations where we want a set of statements to be executed if our variable  $n$  evaluates to one of two values. In this case the code

## Writing Code that Does Stuff

```
case 3:  
case 4: x = 6*n; break;
```

assigns the value  $6*n$  to  $n$  if  $n$  is either 3 or 4.

Finally it is advisable, though not mandatory, to have a `default:` statement at the end of the `switch` evaluation. This statement is executed if none of the `case` statements are true.

# Chapter 5

# The First Complete

# Program

Shown below is a complete program. When programmed into a PIC microcontroller it performed the following functions: when Port B pin 0 was shorted to 5V, it flashed an LED at a rate of once per second. When Port B pin 1 was shorted to 0V, the LED stayed on continuously.

```

/*-----*/
/* Program Name:   LED Flasher           */
/*-----*/

#include <htc.h>

__CONFIG(PROTECT & CPD & LVPDIS & BOREN &
          MCLRDIS & PWRTEN & WDTDIS & INTIO);

void main(void)
{
    unsigned char n, x;

    /* set ports */
    TRISB = 0b00000001;
    PORTB = 0b00000000;

    /* disable comparator */
    CMCON = 0b00000111;

    /* flash LED 5 times */
    while(1)
    {
        if(PORTB & 0b00000001)
        {
            /* turn LED ON for 0.5s */
            for(x=0; x<150; x++)
            {
                for (n=0; n<255; n++)
                {
                    PORTB = 0b00000010;
                }
            }
        }
    }
}

```

## The First Complete Program

```
    }
    /* turn LED OFF for 0.5s */
    for(x=0; x<123; x++)
    {
        for (n=0; n<255; n++)
        {
            PORTB = 0b00000000;
        }
    }
    else
    {
        PORTB = 0b00000010;
    }
}
}
```

Most of the code above we have discussed already. Before we use any variables, we have to declare them to the compiler as well as telling it how much storage space to assign to them. The variables *n* and *x* used in our delay routines are declared with the line

```
unsigned char n, x;
```

The words **unsigned char** are known as a *type definition* and tell the compiler to set aside one byte each of RAM space for variables *n* and *x*. This is explained more in the Data Types chapter. Since most microcontrollers used for embedded applications (and especially the PIC) have little code space, the **unsigned char** data type has been chosen since this only occupies 1 byte.

TRISB and PORTB are defined in our header file, `<htc.h>`. TRISB is the port direction register and that allows us to specify which pins in Port B are inputs and which are outputs. An input is assigned by setting a bit to '1' and an output by setting a bit to '0'. Thus the line

```
TRISB = 0b00000001;
```

sets Port B bit 0 to an input and all other bits in Port B to outputs. We could have equally written

```
TRISB = 0x01;
```

and specified the port values in hexadecimal (0x specifies that the following number is in hex format) but it is easier to see exactly which bits have been set if binary notation is used. See the chapter on Working with Binary Numbers and Bits for a further explanation.

The line

```
PORTB = 0b00000000;
```

sets all of the outputs in Port B to logic '0' or 0V.

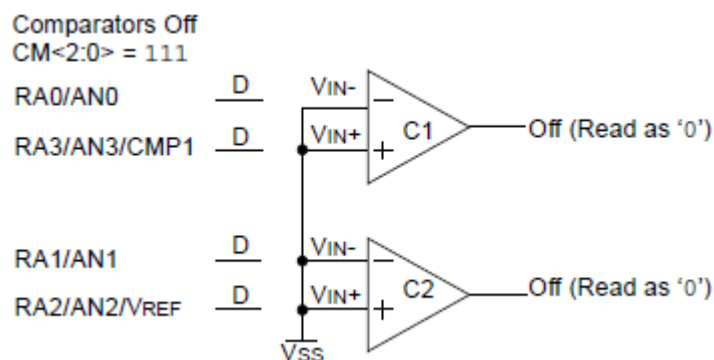
Since we have included `<htc.h>` using the `#include` statement, every time we use `TRISB` and `PORTB` the compiler, it replaces these words with the addresses of the direction register (86hex) and the port register (05hex), thus making our code much easier to understand.

There is a register in the PIC16F627A that controls the internal comparator which we are not using in this case. The register settings for this are shown below

**REGISTER 10-1: CMCON – COMPARATOR CONFIGURATION REGISTER (ADDRESS: 01Fh)**

R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
C2OUT	C1OUT	C2INV	C1INV	CIS	CM2	CM1	CM0
bit 7						bit 0	

We need to disable this comparator as it frees up some port pins. The datasheet tells us how to do this using the picture below



This means we need to set CM<2:0> (this means bits CM2, CM2, CM0) to 111 to disconnect the comparator inputs from the port pins and switch off the outputs. Thus, setting all bits in the CMCON register to 0, except for bits 2,1 and 0 will achieve this. This is accomplished with the line



## The First Complete Program

```
CMCON = 0b00000111;
```

Again, CMCON is defined in our `<htc.h>` file, so we can use the word CMCON instead of its proper address 0x1F.

Once we have disabled the comparator, our next line of code is the start of an infinite loop:

```
while(1)
```

and this will permanently execute the code contained in the curly brackets belonging to the `while` statement. The closing curly bracket for our infinite loop is 24 lines below the opening curly brackets and this is easy to see because the code has been properly indented.

The next line contains the `if` statement

```
if(PORTB & 0b00000001)
```

This line looks at Port B bit 0 and checks to see if it is at logic 0 or logic 1. It does this by using the bitwise AND operator (&). This is explained more in the chapter Working with Binary Numbers and Bits. There are four logical bitwise operators and these perform logical functions on variables. The AND operator compares the corresponding bits of two binary numbers (in this case the binary value read in through Port B and the binary number 0000 0001) and returns either a '1' or '0' depending on the result of each comparison. Only if the corresponding bits are both '1' is a value of '1' returned. Below is an example:

```
      1010 0011
&    1111 0000
-----
      1010 0000
```

In our specific case, if Port B reads back a value of, for example, 1010 0011 and we compare this with a binary value of 0000 0001, we can see that if Port B bit 0 is set to 1 (i.e. shorted to 5V), our `if` statement returns a value '1'. If Port B bit 0 is set to '0' (i.e. shorted to 0V) our `if` statement returns a value of '0'. Moreover, because only bit 0 of our binary number is at '1', when this is ANDed with Port B, bits 1 to 7 of Port B are ignored as these will evaluate to '0' when ANDed with 0000 0001. Thus we have developed an efficient way of inspecting the condition of a single port pin regardless of the state of the other port pins.

## Simple C for the PIC Microcontroller

To be more precise, if the `if` statement returns a *non zero* value it will execute (like the `while` statement mentioned earlier). Thus the `if` statement will execute if Port B bit 0 is shorted to 5V and the LED will be turned ON for 0.5 seconds and OFF for 0.5 seconds. If Port B bit 0 is at 0V, the LED is switched ON and kept ON.

Thus it can be seen that with a few simple C instructions we have written an embedded program for the PIC microcontroller. The next chapter will discuss how to compile this code and program a PIC microcontroller with it.

# **Chapter 6**

# **Simulating Code and**

# **Programming the**

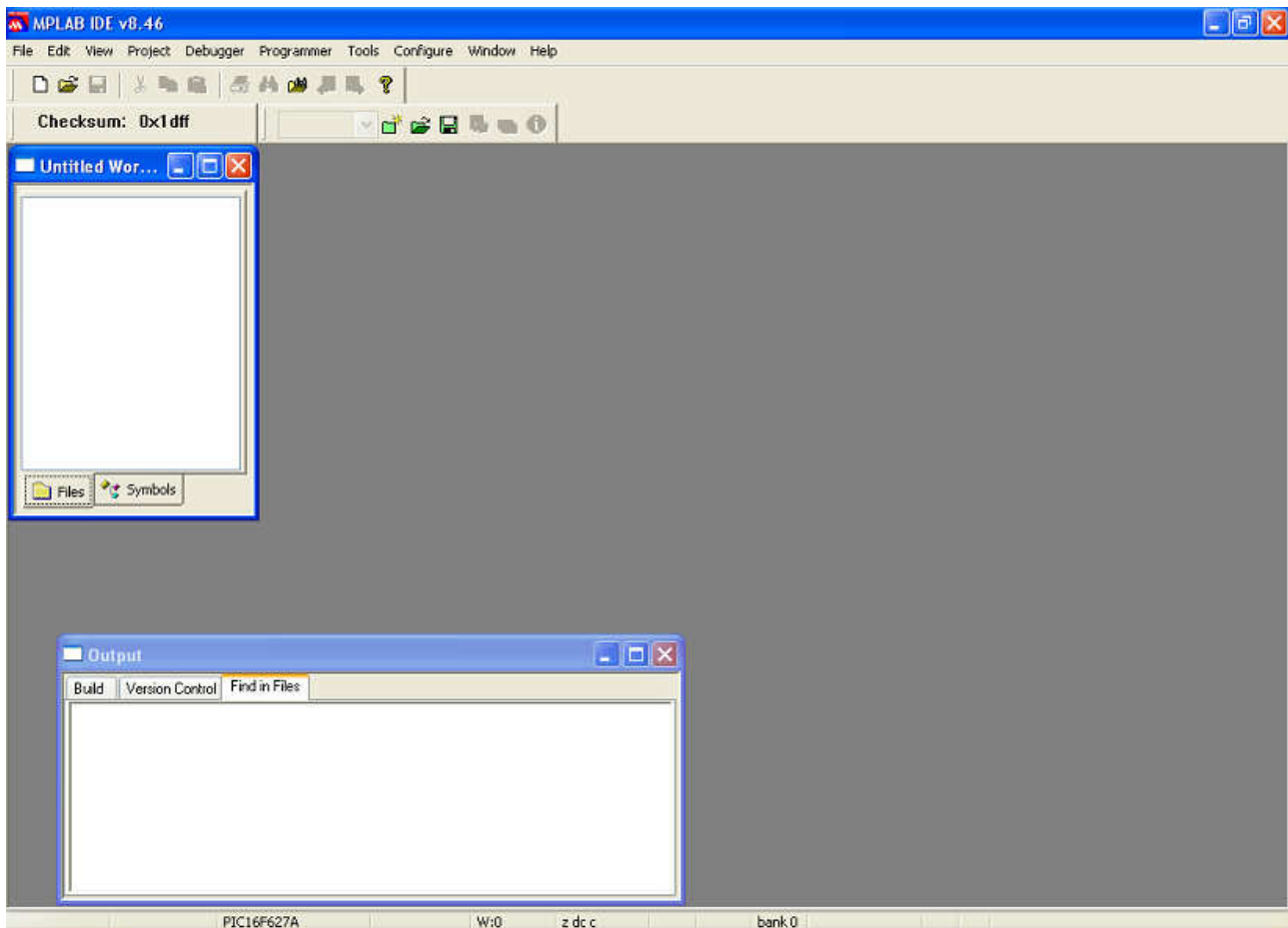
# **PIC**

This chapter will outline the procedure of writing C code in MPLAB, compiling it, simulating it and programming a PIC microcontroller. It will also outline a choice of hardware needed to interface MPLAB with the target PIC device to be programmed.

It is assumed that the Hi Tech editor and MPLAB have been downloaded and installed as outlined in earlier chapters. From the Start button in Windows, drill down to MPLAB IDE and start the program.

The following screen will appear

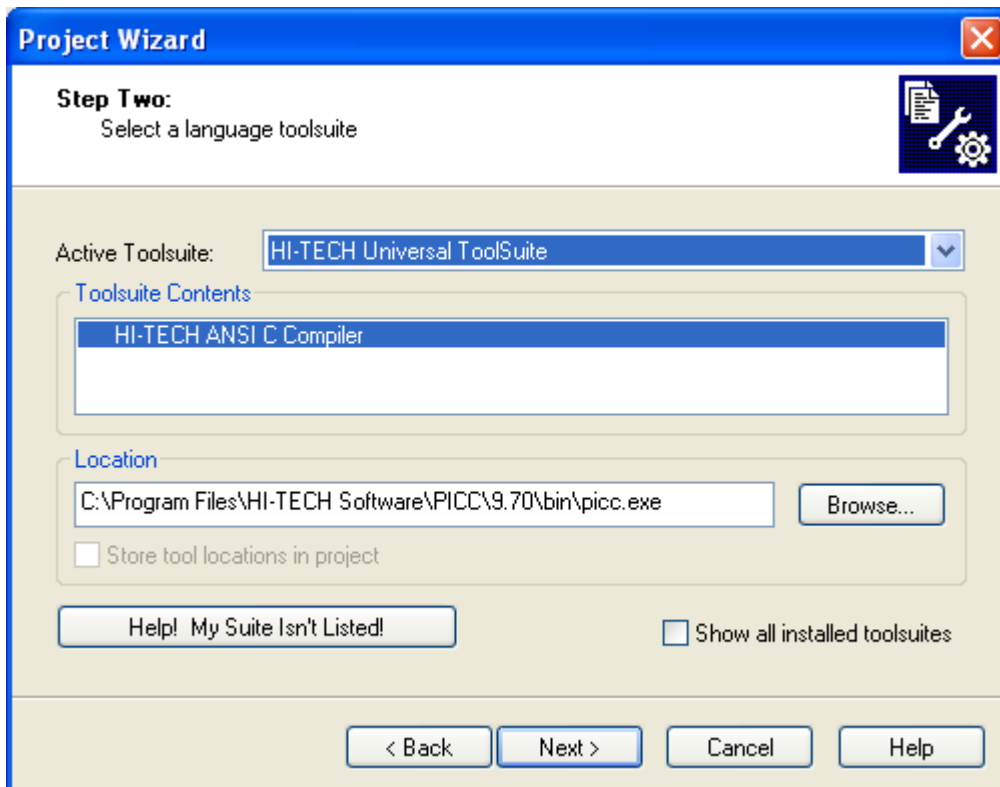
## Simple C for the PIC Microcontroller



In the menu bar, select Project -> Project Wizard. Inside the Project Wizard, select the Device PIC16F627A from the drop down menu then click <Next>.

The next screen is shown below

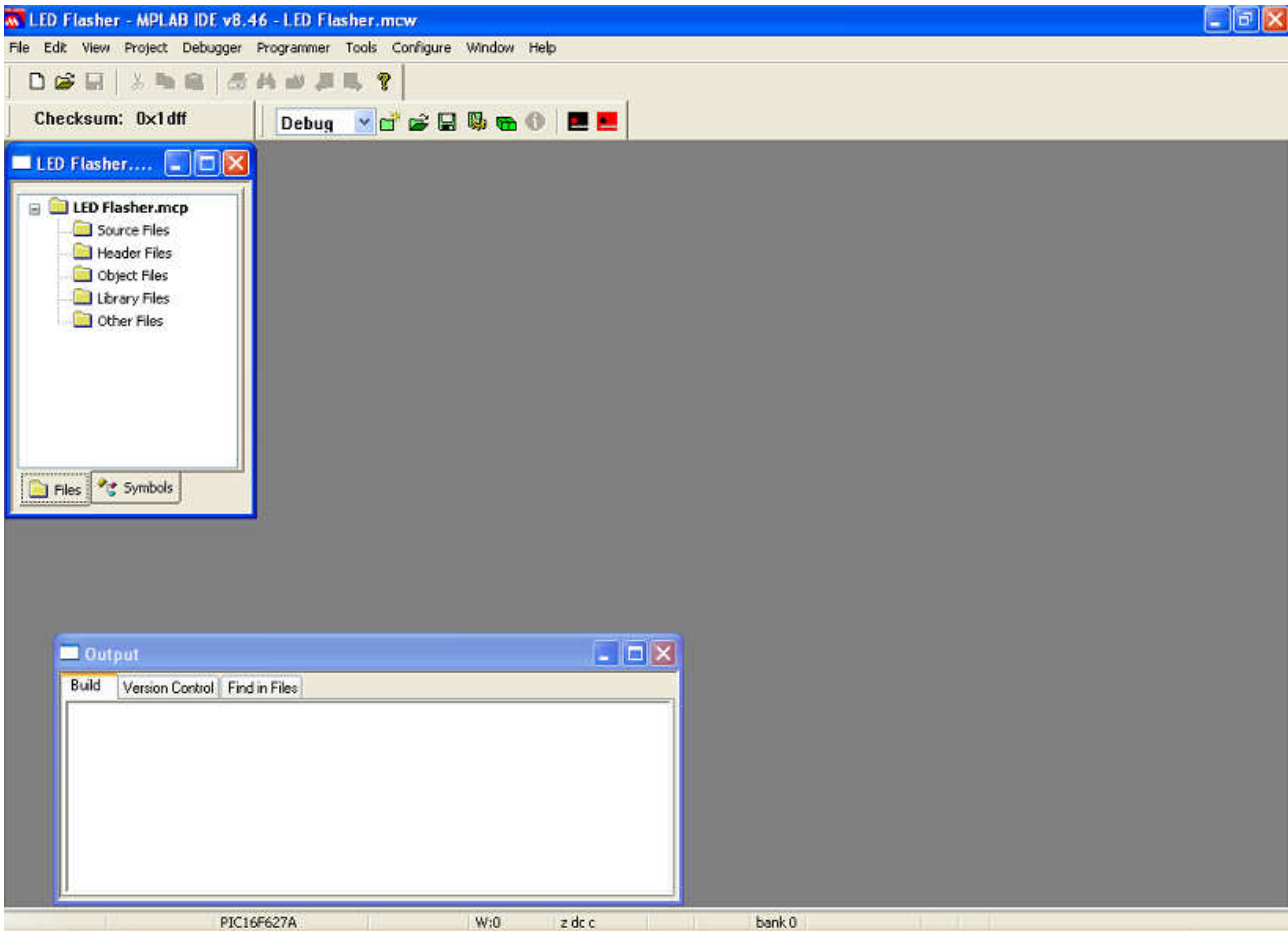
## Simulating Code and Programming the PIC



Select the HI TECH Universal Toolsuite. The Location Box should fill in automatically, but if it does not, browse to the address above (the version above, 9.70, will be different in later installations). Click <Next>.

The next screen asks for a project name and where the project is to be saved. Fill in as necessary, calling the project 'LED Flasher' for example. The next screen asks if any files need to be included in the project. Click <Next> to skip this step. Finally, click <Finish> to complete the project creation. The result is shown below

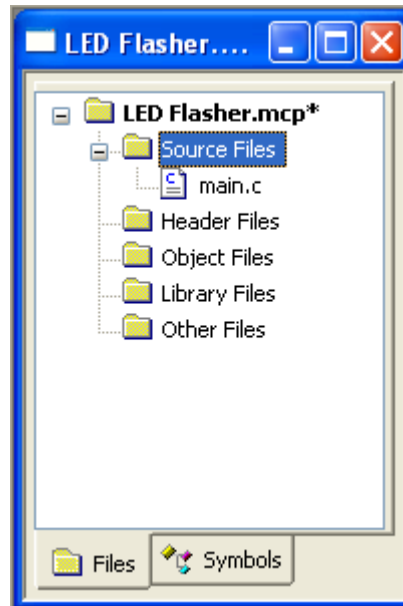
## Simple C for the PIC Microcontroller



We now need to create some files. From the menu bar, select File -> New and this will bring up an untitled blank file. From the menu bar, click File -> Save As and name the file `main.c`. Remember to include the `'c'` as the end as this is going to be the file that stores our C code.

Then in the project box in the top left corner of the screen, right click over 'Source Files' and select the option 'Add Files'. This will bring up a box showing the `main.c` program created earlier. Select this file to add it to our project, thus

## Simulating Code and Programming the PIC

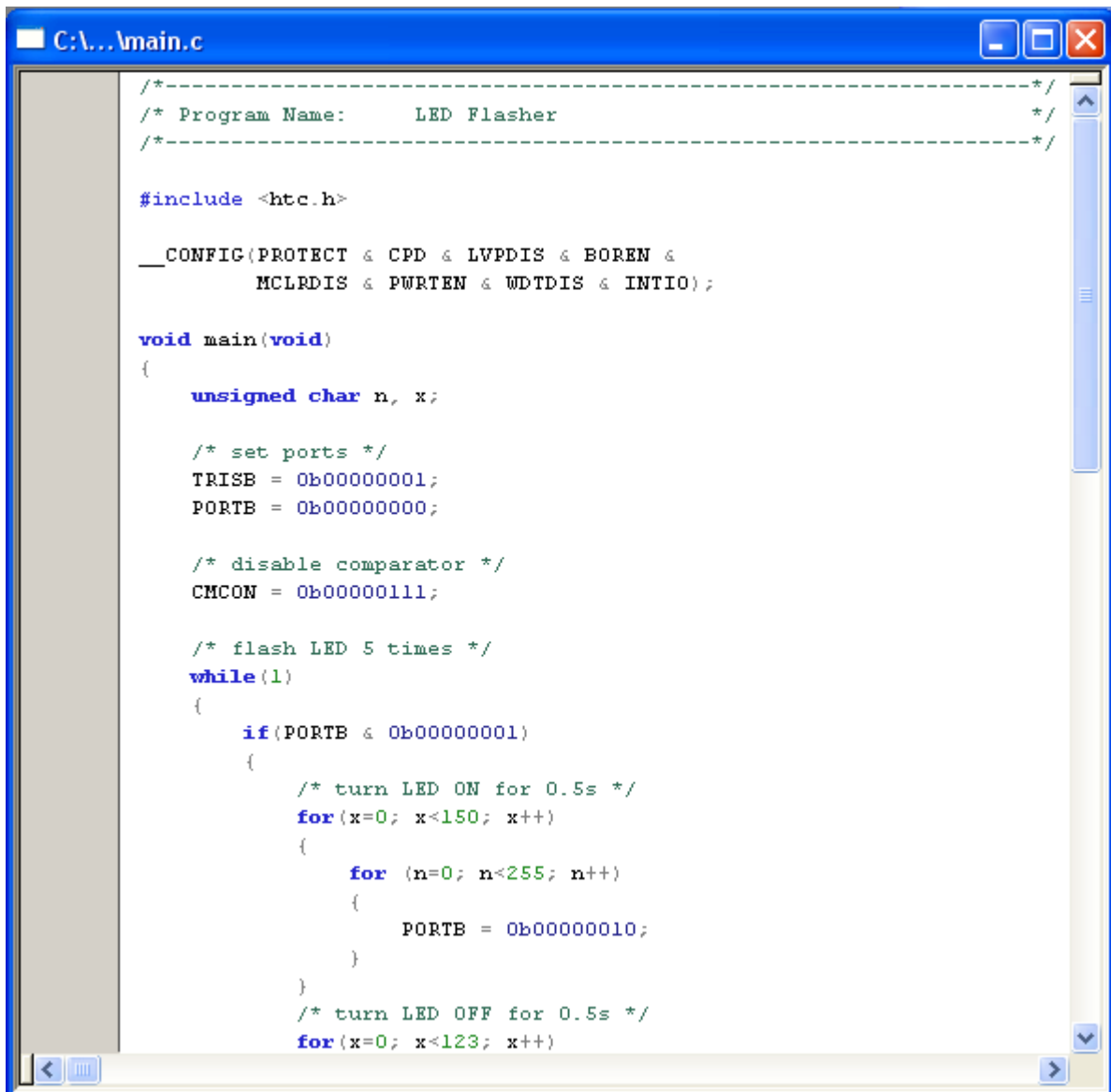


Now, from here it is a matter of personal preference as to whether you use the Hi Tech editor or MPLAB. We prefer the Hi Tech editor as it automatically indents the code, the fonts look nicer etc. If you chose to use the Hi Tech editor, simply open the Hi Tech program by going into the Start button -> Programs -> HI TECH Software -> HI-TIDE v3.15 -> HI-TIDE Integrated Development Environment. Once the program has opened, you can open any of the files created in MPLAB, edit them and save them. If you then go back into MPLAB, it will detect that changes have been made to the file and ask you if you would like to update MPLAB to reflect these changes. Likewise, in MPLAB, if you make any changes then save them, when you go back to the Hi Tech editor, it will detect the changes.

Copy the First Complete Program into the `main.c` file and save it.

Check your typing for semicolon errors and other typing mistakes. Fortunately MPLAB colour codes the text making it easier to spot mistakes (if all your code is in green, you have probably missed off a closing `*/` and MPLAB thinks all of your code is a comment!). Your code should look like

## Simple C for the PIC Microcontroller



```
C:\...\main.c
/*-----*/
/* Program Name:      LED Flasher      */
/*-----*/

#include <htc.h>

__CONFIG(PROTECT & CPD & LVPDIS & BOREN &
          MCLRDIS & PWRTEN & WDTCIS & INTIO);

void main(void)
{
    unsigned char n, x;

    /* set ports */
    TRISB = 0b00000001;
    PORTE = 0b00000000;

    /* disable comparator */
    CMCON = 0b00000111;

    /* flash LED 5 times */
    while(1)
    {
        if(PORTE & 0b00000001)
        {
            /* turn LED ON for 0.5s */
            for(x=0; x<150; x++)
            {
                for (n=0; n<255; n++)
                {
                    PORTE = 0b00000010;
                }
            }
            /* turn LED OFF for 0.5s */
            for(x=0; x<123; x++)

```

We are now going to Build our project. Either hit <F10> or from the menu bar, select Project -> Build. If your code is bug free, you will get the message

```
***** Build successful! *****
```

in the Output Window.

*(note: we noticed a bug in early versions of MPLAB (versions predating v8.50) whereby the \*\*\*\*\* Build successful! \*\*\*\*\* statement did not appear at the end of a successful compilation. This seemed to be a conflict between HI TECH's editor and MPLAB. After installing*



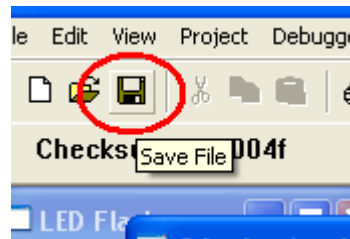
## Simulating Code and Programming the PIC

*MPLAB v8.50, this problem seemed to disappear. We found that if the code was incorrect, the error statements would still appear as expected and if the code was correct, the code compiled and simulated properly and all that was missing was the 'Build successful' statement. )*

If your code has errors, you will get the message

\*\*\*\*\* Build failed! \*\*\*\*\*

Warning: If your build fails and you re-Build your project without correcting the error, you might get a 'Build Successful' message. Only build your project when the Save Icon is NOT 'greyed out' thus



If you have build errors, they will look something like this:

```
Error [312] C:\Documents and Settings\My Documents\LED Flasher\main.c; 32.1 ";"  
expected
```

This is telling us the error code (312), the fact that the error is in `main.c` and on line 32 of the file. It then goes on to say that a semicolon is missing. In fact, the error is actually on line 31, but the compiler only realised there was an error when it got to line 32 and realised the semicolon from line 31 was missing. Double clicking on the error message will take us to the suspected error.

If you get any other error message, the compiler manual has a list of them and a friendly explanation for each one. This can be found in

`C:\Program Files\HI-TECH Software\PICC\9.70\docs>manual.pdf`.

Correct the bugs until you have code that compiles OK.

If your code still does not work, it might be worth going back to the original template program describe at the start of this book and compiling that, then adding functions one by one until the error appears.

We now need to simulate our code. This is far easier than programming a part, plugging it into a target board and wondering why nothing happens, so it is wise to get into the habit of simulating our code every time a change has been made until we are certain that our code works perfectly.

Firstly, we need to set up the microprocessor speed. We discussed earlier about generating delay routines to flash our LED on and off. Setting the microprocessor speed inside the simulator allows

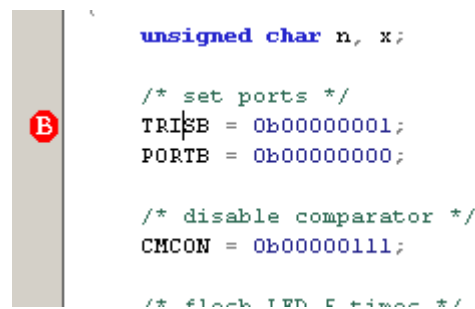
## Simple C for the PIC Microcontroller

us to see exactly how long each line of code takes to execute. From the menu bar, select Debugger -> Settings, go to the Osc/Trace tab and in the Processor Frequency box, type in 4. This sets MPLAB to simulate our code as though it were running on a processor with a 4MHz clock. Click OK to close the dialogue box.

To simulate the code, from the menu bar, select Debugger -> Select Tool -> MPLAB SIM. We are now going to Single Step through the code and inspect how the ports and registers change as each line of code executes.

Now, our compiler generates some startup code for us, written in assembly language that we don't really need to worry about. Once the microprocessor is reset, Single Stepping will take us through this assembly code. It is advisable to skip over it by setting a *breakpoint* in our code. A breakpoint allows the simulator to run through the code, uninhibited until it reaches the breakpoint, where it stops. Breakpoints are extremely useful because they allow our simulator to execute code in near real-time without us having to single step through every line of code.

Set a breakpoint by double clicking in the grey area next to the TRISB statement



```
unsigned char n, x;

/* set ports */
TRISB = 0b00000001;
PORTB = 0b00000000;

/* disable comparator */
CMCON = 0b00000111;

/* flash LED 5 times */
```

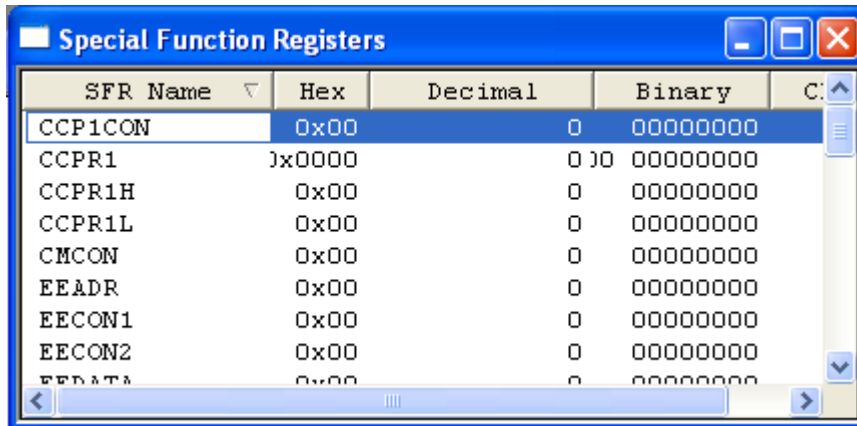
The image shows a code editor window with a vertical grey bar on the left side. A red circle with a white letter 'B' is positioned on the grey bar, aligned with the line of code: `TRISB = 0b00000001;`. This indicates that a breakpoint has been set at this line of code.

Reset the microprocessor by hitting the <F6> key then execute the code up to the breakpoint by hitting the <F9> key. Hitting the <F9> key runs our code up to a breakpoint, so we have neatly skipped over all the startup assembly code and landed at the point in code that is just about to set the TRIS register. Your breakpoint should now have a green arrow over it telling us that our code has executed up to this line of code.

Hitting <F7> thereafter single steps through the code, line by line and changes the registers to show us what is happening. We now need to view these registers.

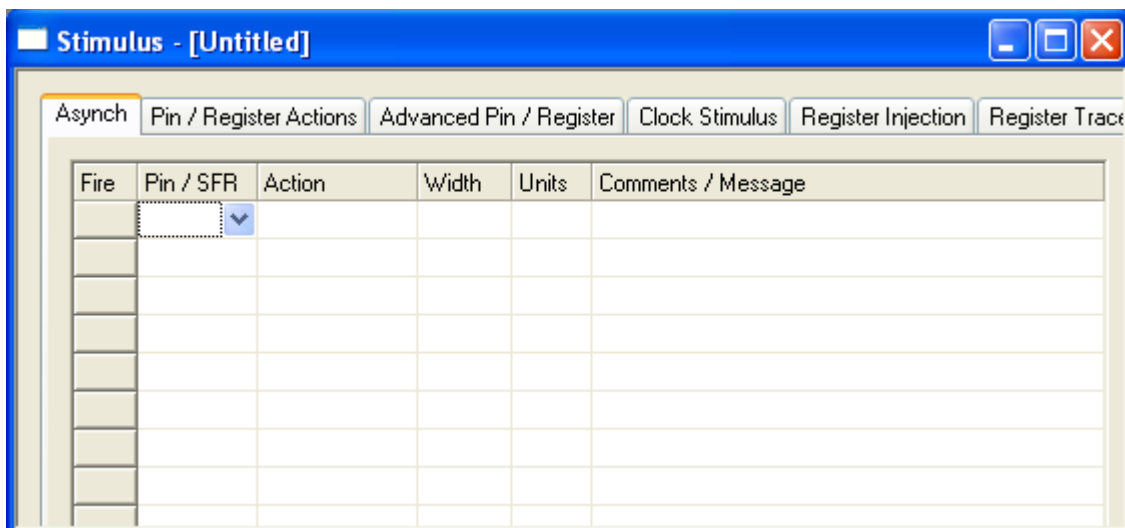
From the menu bar, select View -> Special Function Registers and the following window will appear.

## Simulating Code and Programming the PIC



Scrolling down this view we can see all the registers inside the microcontroller. The registers that have just changed (after the execution of the last instruction) are shown in red. Scroll down to TRISB, click the mouse in the `main.c` window, hit <F7> and if TRISB changes, the colour of the TRISB line will change to red. Regardless, once this line of code has been executed, TRISB should have a value of 0000 0001. Hitting <F7> repeatedly allows us to single step through our code, watching PORTB, CMCON etc change.

Now somehow we need to be able to apply an input to simulate a port pin being taken high (to 5V) or low (to 0V). From the menu bar, select Debugger -> Stimulus -> New Workbook and the following window will appear



Under the Pin/SFR heading, select RB0, under Action select 'Set Low' and under the Comments/Message heading write 'Input Pin'. Under the Fire heading an arrow should have appeared. Click this arrow to set RB0 Low.

Reset the microcontroller (<F6>), hit <F9> to run to the breakpoint then hit <F7> to single step through the code.

Single step to the line

```
if(PORTB & 0b00000001)
```

Go to the Special Function Registers window and find PORT B. Single stepping past this point takes us to the line

```
PORTB = 0b00000010;
```

Showing that the `if` statement has evaluated untrue and we have skipped all the commands inside the `if` curly brackets. Once this line is executed, we notice that Port B bit 1 has been set, as expected. Continued single stepping causes the program to hop between the `if` evaluation and the `PORTB = 0b00000010` assignment.

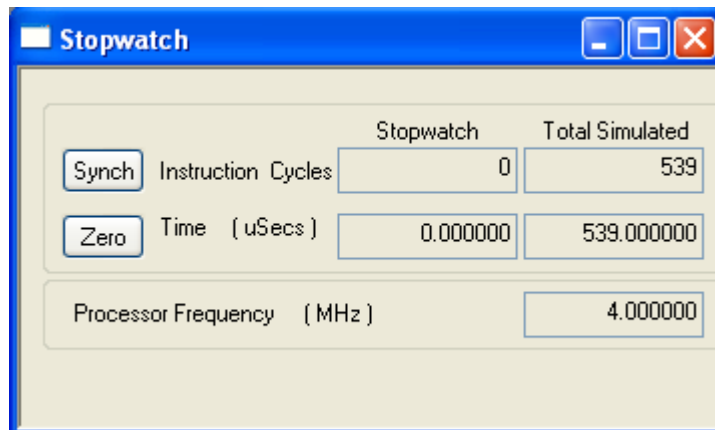
Now go back to the stimulus window and under Action, set RB0 High. Hit the Fire button and continue to single step through the code. We can now see that the `if` statement is executed and we enter the nested `for` loop.

We now need to inspect the counter  $n$  inside the inner `for` loop. From the menu bar, select View -> Locals to bring up a window showing the variables  $x$  and  $n$ . As we single step through the code, we see  $n$  increments while  $x$  remains at 0.

Inside the Locals window, we can double click on the value of  $n$  and change it. Make sure one of the headings inside this window reads 'Decimal' to show our value of  $n$  in decimal. This can be done by right clicking over the heading and ensuring 'Decimal' is checked. Change the value of  $n$  to 250. A few more single steps through the code should increment  $n$  over 255, the inner ( $n$ ) loop will stop, the outer ( $x$ ) loop will increment by one and the inner loop starts again.

We are now going to test how long our LED is ON and how long it is OFF. From the menu bar, select Debugger -> Stopwatch thus

## Simulating Code and Programming the PIC



Double click next to the line

```
PORTB = 0b00000000;
```

to set a breakpoint here. This is the line of code that clears Port B and switches OFF the LED. Hit <F9> to run to that breakpoint. You will have noticed that the Stopwatch has changed its value and now displays how long it has taken to get to the breakpoint. Click on the 'Zero' button to clear the timer.

Locate the line

```
PORTB = 0b00000010;
```

This is the line inside the **if** statement, NOT the one inside the **else** statement. Double click next to this line to set a breakpoint here. This line sets Port B bit 1 and turns ON our LED.

Double click on the breakpoint next to the line

```
PORTB = 0b00000000;
```

to *clear* the breakpoint. Now hitting <F9> will take us to the line of code that switches on the LED. Inspecting the timer shows that the code has taken 503ms to execute, so our LED has been off for 503ms. Clear the timer again. Set a breakpoint next to the line

```
PORTB = 0b00000000;
```

clear the breakpoint next to the line

```
PORTB = 0b00000010;
```

Hit <F9> again and we see that the time the LED is ON is 499ms, so we can see that our LED is now flashing ON for 503ms and OFF for 499ms.

Our code has now been compiled and simulated. Now we need to program it into the microprocessor. The following procedures will vary depending on what programmer you have. Below will be described the procedure for the PICSTART Plus.

### Programming a PIC

Now some basic code has been written, it is time to program a microcontroller. There are a whole myriad of programmers available to the engineer on a restricted budget. The author has always used the PICSTART Plus from Microchip, although at time of writing, this was priced at £125, so might be beyond the budget of some.

Velleman Electronics have a programmer (serial number K8048) with a simple interface at a much lower cost that can be purchased from a variety of stockists globally. This can be investigated further at:

[www.velleman.be](http://www.velleman.be)

We have also discovered an article in EPE mag ([www.epemag.com](http://www.epemag.com)) UK edition, May 2010 that is a low cost programmer for PICs and dsPICs.

The two programmers above have not been tested by us though.

In addition, because of the popularity of the PIC microcontroller, there are hundreds of designs available on the web that allow someone with only limited soldering skills to build a programmer.

The rest of this chapter will be dedicated to the PICSTART Plus programmer as this has performed fault free for over 10 years.

Connect the PICSTART Plus to the RS 232 port of your PC. Connect the power to the PICSTART Plus. From the menu bar, select Programmer -> Select Programmer then choose PICSTART Plus. Then under Programmer -> Settings in the Communications tab, select the COM port that your

## Simulating Code and Programming the PIC

PICSTART Plus is connected to. Click OK. Now select Programmer -> Enable Programmer. A message should come up in the Output window showing

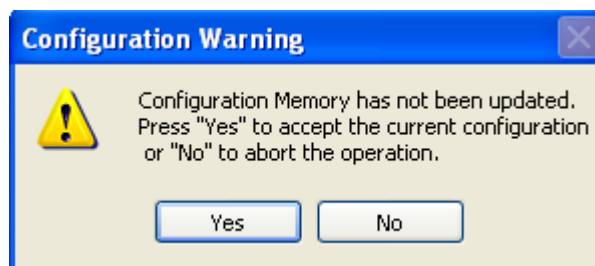
PICSTART Plus Firmware Version 04.50.11

Place the PIC16F627A into the PICSTART Plus socket lining up Pin 1. Close the handle down to clamp the chip. In MPLAB, from the menu bar select Programmer -> Erase Flash Device. The Output window should display

Erase operation is successful.

Then select Programmer -> Program to program the chip. During the Build process, the compiler produces a .hex file. This file is written in machine code and can be understood by the microprocessor. It is this hex file that is downloaded to the chip.

If you get a warning



select 'Yes' to accept the current configuration. Once the device has been programmed successfully, you will get the message:

Programming/Verification completed successfully

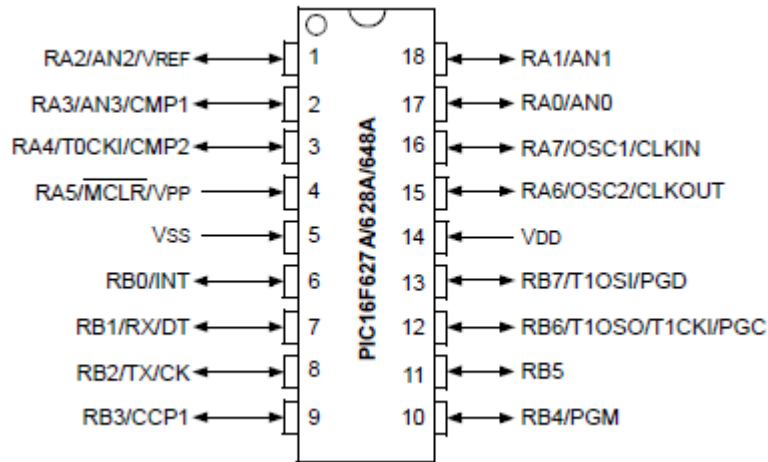
If any warnings appear, check that you have remembered to 'Erase Flash Device', check that the chip is seated in the programmer correctly and the power supply is connected to the PICSTART Plus.

Warning:

If you want to go back and resimulate your code, you have to go back to the menu bar and select Debugger -> Select Tool -> MPLAB SIM. Likewise, once in the simulator mode, to get back into programming mode, you need to go back to the menu bar and select Programmer -> Select Programmer -> PICSTART Plus

Once your PIC is programmed, you need to put it into a circuit. The good thing about PICs is that they require no external components to work. From the figure below, we can see that we need to connect 5V to Vdd (pin 14) and 0V to Vss (pin 5).

## Simple C for the PIC Microcontroller



We need to connect a jumper to Port B pin 0 and an LED and resistor to Port B bit 1. A suitable circuit is shown in Appendix D: Suggested LED Schematics.

Nothing more is needed! Just check the power supply is connected correctly and switch on. The LED should flash when Port B bit 0 is connected to the 5V supply and stay on permanently when Port B bit 0 is connected to 0V.



# Chapter 7

# Timers and

# Interrupts

Although the code in the previous chapter does exactly what we want it to do, there is a neater way of achieving the same goal. Most PICs have one or more timers enabling us to create delay routines that run in the background. They tick away incrementing every 4 clock cycles, regardless of what the rest of the code is doing. This enables us to create very precise timing delays without relying on lines of code to make up the delay.

Most PICs also include an interrupt structure that allows program code to be diverted when certain events happen. Such events include Timer Overflow, Change on Port Pin, Data Received on the UART etc. While intimidating to begin with, learning how to harness the power of the interrupt will make your code easier to write and understand and give your programs much more flexibility.

Therefore this chapter is going to examine how to combine Timers with Interrupts to modify our LED Flasher program.

The PIC16F627A has 3 timers: two 8-bit timers (Timer 0 and Timer 2) and a 16 bit timer (Timer 1). The 8 bit timers count from 0 to 255 ( $2^8 - 1$ ) then 'overflow' back to 0. The 16 bit timer counts from 0 to 65535 ( $2^{16} - 1$ ) then overflows back to 0. When the overflow occurs, an interrupt is generated, telling us that our delay has passed.

We are going to use the 16 bit timer. Firstly we need to configure the timer. The datasheets shows us that the control register for Timer 1 is

**REGISTER 7-1: T1CON – TIMER1 CONTROL REGISTER (ADDRESS: 10h)**

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	
—	—	T1CKPS1	T1CKPS0	T1OSCEN	$\overline{T1SYNC}$	TMR1CS	TMR1ON	
bit 7								bit 0

It is worthwhile locating this register in the datasheet and reading the explanations of each bit. The above figure not only tells us the names of the bits inside the register, but it also tells us if they are

Unused (U), Read Only (R), Write Only (W) or Read/Write (R/W). It also tells us their state on power-up and we can see that all of the bits in the register above are reset to '0' on power-up.

These will be expanded on below.

Bits 7 and 6 are unused, so we are going to set them equal to zero.

Bits 5 and 4 are T1CKPS1 and T1CKPS0 which the datasheet tells us are the Timer 1 clock prescale bits. Normally Timer 1 increments every instruction cycle. Since we are using the internal 4MHz clock our *clock cycle* is 0.25us long. There are 4 clock cycles per instruction cycle so our *instruction cycle* is 1us long. In our LED flasher program, we need to generate a delay of 0.5 seconds. Timer 1 can count to a maximum of 65535 and this equates to a delay time of (65535 x 1us = 65.535ms). This is clearly not long enough. The prescaler allows us to change the time between increments of Timer 1 and the datasheet tells us we can change this from 1:1 (no change) to 1:8 (Timer 1 increments every 8 instruction cycles, or 8us).

If Timer 1 increments every 8us and can count to 65535, this allows us to count to 0.52428 seconds (8us x 65535), which means we can achieve a delay of over 0.5 seconds with a prescaler of 1:8.

To calculate what Timer 1 has to count up to for a delay of 0.5 seconds, we have to work backwards.

$$count = \frac{0.5}{8us} = 62500$$

Now, we want to make use of our interrupts. Interrupts only occur when Timer 1 overflows. It would be much nicer if we could generate an interrupt when the counter reaches 62500, but PICs aren't built that way!

We therefore need to *preload* Timer 1 with a value such that it will overflow after the desired time. We know that Timer 1 can count to 65535, so the value we preload Timer 1 with is

$$(65536 - 62500) = 3036$$

Thus, Timer 1 counts from 3036 to 65536 (62500 counts) then generates an interrupt when it overflows. We will see how to do this later. The datasheet says that to set the prescaler equal to 1:8 we need to set T1CKPS1 to '1' and T1CKPS0 to '1'.

Bit 3 is T1OSCEN, or Timer 1 oscillator enable control bit. This is an internal oscillator that can be connected up to an external 32.768kHz crystal for timing seconds, minutes, hours etc. We are going to use the main 4MHz oscillator, so we set this bit to '0'.

Bit 2 is T1SYNC allowing us to synchronise the incrementing of Timer 1 with an external clock. We are not going to do this and the datasheet tells us to set this bit to '1' if we do not need external

## Timers and Interrupts

synchronisation. Incidentally, T1SYNC has a bar above it in the datasheet – see the Register diagram above. This means that its function is activated by setting this bit equal to a ‘0’ instead of a ‘1’ (it is ‘active low’ not ‘active high’). This can sometimes also be written as  $\overline{\text{T1SYNC}}$  with a forward slash before it. Therefore, if we wanted to activate the synchronisation of Timer 1 we would set Bit 2 equal to logic ‘0’.

Bit 1 is TMR1CS and is the Timer 1 clock source bit. We want our clock source to be the internal 4MHz clock and the datasheet tells us to set this bit to ‘0’ to use the internal clock.

Bit 0 is TMR1ON which is the ON/OFF bit of Timer 1. We don’t want Timer 1 to start just yet, so we will set this bit later. For the moment, to turn Timer 1 off, we set Bit 0 to ‘0’.

All of the above can be configured neatly in one line thus

```
T1CON = 0b00110100;
```

This means our Timer 1 increments using the internal clock (multiplied by 8) as its clock source, increments every 8 $\mu$ s and it is currently turned off. We have written the T1CON value in binary as it makes it easier to see which bits are set and which bits are cleared.

Now we need to preload the Timer 1 registers to ensure we get an overflow after 62500 counts.

Timer 1 has two 8 bit registers to hold the 16 bit count value. These are TMR1L (Timer 1 low) and TMR1H (Timer 1 high). We need to preload these registers with the *decimal* value 3036. We are going to use hex numbers to load TMR1L and TMR1H as it is easier to work out what number has to go in TMR1L and what number has to go into TMR1H.

The Calculator function in Windows (in scientific mode) can be used to convert numbers between binary, decimal and hex. It can be started from the Start Button -> Programs -> Accessories -> Calculator. We can see that 3036 decimal is 0BDC $_{hex}$ . The chapter entitled Working with Binary Numbers and Bits explains that each hex digit is 4 binary bits and therefore a 16 bit value can be represented by 4 hex digits. Therefore we can immediately see that the upper two hex digits (0B) need to be loaded into TMR1H and the lower two hex digits (DC) need to be loaded into TMR1L. This can be achieved with the following lines of code:

```
TMR1H = 0x0B;  
TMR1L = 0xDC;
```

Our timer is now configured. We now need to configure the interrupts to ensure we get an interrupt when Timer 1 overflows. The datasheet has a handy table that shows us what registers to configure to use Timer 1.

**TABLE 7-2: REGISTERS ASSOCIATED WITH TIMER1 AS A TIMER/COUNTER**

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR	Value on all other Resets
0Bh, 8Bh, 10Bh, 18Bh	INTCON	GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	0000 000u
0Ch	PIR1	EEIF	CMIF	RCIF	TXIF	—	CCP1IF	TMR2IF	TMR1IF	0000 -000	0000 -000
8Ch	PIE1	EEIE	CMIE	RCIE	TXIE	—	CCP1IE	TMR2IE	TMR1IE	0000 -000	0000 -000
0Eh	TMR1L	Holding Register for the Least Significant Byte of the 16-bit TMR1 Register								xxxxx xxxxx	uuuuu uuuuu
0Fh	TMR1H	Holding Register for the Most Significant Byte of the 16-bit TMR1 Register								xxxxx xxxxx	uuuuu uuuuu
10h	T1CON	—	—	T1CKPS1	T1CKPS0	T1OSCEN	T1SYNC	TMR1CS	TMR1ON	--00 0000	--uu uuuuu

Legend: x = unknown, u = unchanged, - = unimplemented read as '0'. Shaded cells are not used by the Timer1 module.

We have configured TMR1L, TMR1H and T1CON, so we will now address INTCON, PIR1 and PIE1 below.

The INTCON register is shown below

**REGISTER 4-3: INTCON – INTERRUPT CONTROL REGISTER (ADDRESS: 0Bh, 8Bh, 10Bh, 18Bh)**

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x	
GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	
bit 7								bit 0

Bit 7 is GIE and is the global interrupt bit. If this is cleared, whatever we set any of the other bits to will have no effect, so this needs to be set to '1'

Bit 6 is PEIE and is the Peripheral Interrupt Enable bit. Timer 1 is classed as a peripheral (we know this because Timer 1 is mentioned in the Peripheral Interrupt Enable Register – see below). This bit therefore needs to be set to '1'

Bit 5 is T0IE and is the interrupt enable bit for Timer 0. We are not using this timer, so can set this bit to '0'

Bit 4 is INTE and enables us to use Port RB0 as an external interrupt input. We do not use this, so set this bit to '0'

Bit 3 is RBIE and enables us to use a logic level change on any of Port B's pins to act as an interrupt. We do not use this, so set this bit to '0'

Bit 2 is TMR0 and tells us that an interrupt has occurred on Timer 0. We set this to '0' as we are not using Timer 0.

Bit 1 is INTF and tells us that an external interrupt has been generated. We set this to '0' as we are not using external interrupts

## Timers and Interrupts

Bit 0 is RBIF and tells us that there has been a change on Port B. We are not using the ‘Port B change’ interrupt so set this to ‘0’.

We can set the INTCON register to the above settings with the line:

```
INTCON = 0b11000000;
```

Now we need to configure the PIE1 register

### REGISTER 4-4: PIE1 – PERIPHERAL INTERRUPT ENABLE REGISTER 1 (ADDRESS: 8Ch)

R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0	R/W-0
EEIE	CMIE	RCIE	TXIE	—	CCP1IE	TMR2IE	TMR1IE
bit 7							bit 0

The only bit we need to worry about in PIE1 is bit 0. All other bits are not related to Timer 1. Bit 0 is TMR1IE, the Timer 1 Interrupt Enable bit. Set this bit to ‘1’

We can set the PIE1 register to the above settings with the line:

```
PIE1 = 0b00000001;
```

Finally, we need to configure the PIR1 register.

### REGISTER 4-5: PIR1 – PERIPHERAL INTERRUPT REGISTER 1 (ADDRESS: 0Ch)

R/W-0	R/W-0	R-0	R-0	U-0	R/W-0	R/W-0	R/W-0
EEIF	CMIF	RCIF	TXIF	—	CCP1IF	TMR2IF	TMR1IF
bit 7							bit 0

Now, the only bit applicable to Timer 1 is the TMR1IF flag. This *should* be reset to ‘0’, but just in case it is not we are going to reset it using the line

```
PIR1 = 0b00000000;
```

We now need to write the Interrupt Service Routine (ISR). The Hi Tech compiler manual has a reserved word `interrupt` to tell the compiler that the function you are just about to write is an

interrupt. The compiler then makes sure that the appropriate registers are saved before the interrupt is serviced. A simple ISR is written below:

```
void interrupt isr(void)                                /* int service routine */
{
    if (TMR1IF == 1)
    {
        TMR1IF = 0;
        global = 0x01;                                /* indicate overflow */
        TMR1H = 0x0B;
        TMR1L = 0xDC;
    }
}
```

The above code is only executed when an interrupt occurs. It checks the TMR1IF flag to see if the interrupt has been generated because of an overflow on Timer 1. If this is the case, it resets the flag (to stop any further interrupts) and sets bit 0 of a general purpose register called 'global'. This is a register we are going to define in our code and is a useful way of indicating that certain events have occurred. After this, because we want another interrupt to be generated in 0.5 seconds time, here is a good place to reload our timer values in TMR1H and TMR1L.

Here is the complete code:

```
/*-----*/
/* Program Name:   LED Flasher                               */
/*-----*/

#include <htc.h>

__CONFIG(PROTECT & CPD & LVPDIS & BOREN &
         MCLRDIS & PWRTEN & WDTDIS & INTIO);

unsigned char global = 0;

/*-----*/
/* INTERRUPTS                                             */
/*-----*/
void interrupt isr(void)                                /* int service routine */
{
    if (TMR1IF == 1)
    {
        TMR1IF = 0;
        global = 0x01;                                /* indicate overflow */
        TMR1H = 0x0B;
        TMR1L = 0xDC;
    }
}
```

## Timers and Interrupts

```
/*-----*/
/* MAIN PROGRAM */
/*-----*/
void main(void)
{
    /* set ports */
    TRISB = 0b00000001;
    PORTB = 0b00000000;

    /* disable comparator */
    CMCON = 0b00000111;

    /* initialise Timer 1 */
    T1CON = 0b00110100;
    TMR1H = 0x0B;
    TMR1L = 0xDC;

    /* initialise interrupts */
    INTCON = 0b11000000;
    PIE1 = 0b00000001;
    PIR1 = 0b00000000;

    TMR1ON = 1;

    /* flash LED 5 times */
    while(1)
    {
        if(PORTB & 0b00000001)
        {
            /* turn LED ON for 0.5s */
            while((global & 0b00000001)==0)
            {
                PORTB = 0b00000010;
            }
            global = global & 0b11111110;

            /* turn LED OFF for 0.5s */
            while((global & 0b00000001)==0)
            {
                PORTB = 0b00000000;
            }
            global = global & 0b11111110;
        }
        else
        {
            PORTB = 0b00000010;
        }
    }
}
```

There are 2 functions:

```
void interrupt isr(void)    and    void main(void)
```

Any variables defined within a function can only be used inside that function whereas variables defined outside of the functions can be used by all functions. In our First Complete Program, we implemented the line

```
unsigned char n, x;
```

as we wanted to use variables *n* and *x* as counters for the loops. If we had other functions in our program, we could equally use the same variables names *n* and *x*, but if they are defined in separate functions, they would be completely different variables. In other words, the variable *n* could be defined in more than one function and modifying *n* in one function will not affect the variable *n* in another function. However, if the variables are defined outside the functions, then the variables are *global*. They can be used across all functions. This is what we are doing with the line

```
unsigned char global = 0;
```

Since the variable `global` is defined outside of `interrupt isr` and `main` it can be used by both functions.

This allows us to set bit 0 of the `global` variable inside the `interrupt` function, then inspect its value in the `main` routine. This is explained more in the Functions chapter.

All the registers associated with Timer 1 are programmed as described in the text above. We also opted to start Timer 1 just before we used it. This is done using the instruction

```
TMR1ON = 1;
```

`TMR1ON` is the Timer 1 ON bit inside the `T1CON` register and is defined in our header file, `htc.h`. Quite handily, the header file given to us by Hi Tech also includes a vast array of bit definitions as well as register definitions. We could equally write to the entire `T1CON` register, but the above instruction allows us to individually address the `TMR1ON` bit without affecting the other bits.

This project can be set up in MPLAB as before and the code can be single stepped and the registers can be seen in the Special Function Registers window. A stimulus file can be set up to set Port B bit 0 to logic '1' or logic '0'. If you want to inspect the `global` register, this can be seen by going into



## Timers and Interrupts

the menu bar, clicking on View -> Watch then inside this window, click on the drop down list next to the <Add Symbol> button.

The code scrolls through the 'LED ON' phase using the following loop

```
while((global & 0b00000001)==0)
{
    PORTB = 0b00000010;
}
```

until an interrupt occurs. As it loops through the above code, the Special Functions Registers window allows us to see Timer 1 incrementing. Here, we are just constantly setting Port B bit 1 to '1', but we could do something far more useful and leave Timer 1 to increment in the background.

When TMR1L and TMR1H roll over to 0000 0000, the TMR1IF flag is set in the PIR1 register causing the interrupt to assert and the code jumps to our interrupt service routine.

Inside the ISR, the TMR1IF flag is reset, the Timer is reloaded with the hex value OBDC and bit 0 of the `global` register is set to 1. The code then returns to its pre-interrupt address and continues to execute. We see that the loop evaluating bit 0 of the `global` register is now false, so the loop is exited, bit 0 in the `global` register is set to '0' and the code moves to the 'LED OFF' phase. For more information on how to set and clear bit, refer to the chapter on Working with Binary Numbers and Bits.

Finally, testing the code in MPLAB using the stopwatch, shows us that the ON time is 500.044ms and the OFF time is 500.034ms, so we have gained precision in our timing (because our timer increments in shorter intervals than a loop in our previous code) as well as enabled the timer to run in the background while our code can do something else.

Programming the above code into a PIC16F627A causes the LED connected to Port B bit 1 to flash (ON for 0.5 seconds, OFF for 0.5 seconds) if Port B bit 0 is held high. If Port B is connected to ground, the LED stays permanently on.

# Chapter 8

# Functions

Further improvement can be made to the readability of our code. It is desirable to write the principal block of our code (in the `main` program) as nothing more than a series of ‘calls’ to other function blocks each with a specific tasks to execute. Using the example above, our `main` function might consist of calls to functions blocks to execute, say, the micro initialisation, LED ‘flash’ routine and LED ‘permanently ON’ routine. Our program (in pseudo code) would then look something like

```
void main(void)
{
    /* initialise micro */
    Call initialisation routine

    TMR1ON = 1;

    /* flash LED 5 times */
    while(1)
    {
        if(PORTB & 0b00000001)
        {
            /* flash LED on and off */
            Call LED flash routine
        }
        else
        {
            Call LED ON routine
        }
    }
}
```

In the above code, we have moved the individual lines of code into separate files and these files are then called in the `main` program. Writing code as a series of separate functions makes code a lot easier to debug and much easier to read. If you have a problem with the LED flashing routine, this problem will manifest itself *every* time that routine is called. The LED flash routine could be tested as an individual entity and once we know that it works, it can be merely treated as a ‘black box’ that executes a desired task without having to re-analyse how it does it.

## Functions

Moreover, if later a program requires an LED flash routine, the file containing the code can be copied over, knowing it works, and easily imported into the other program. Writing programs as a series of individual functions (called *structured programming*) is definitely the way to go.

As programs grow in complexity, thousands of lines of code can be broken down into a series of simpler functions and this is another benefit of using a structured programming approach. Sometimes it is nearly impossible, once given a design to implement in software, knowing where to start. If a data logger has to be designed, we can immediately assume that we will need an LCD display, an interface to download data, a user interface to read in button presses, a power management routine that might switch on and off certain parts of circuitry. It is often to mentally easier to start writing either easy code, or code that are you familiar with. Implementing structured programming enables us to do exactly this, then move on the more complex functions, then write some 'glue' code to bond all the functions together in our `main` function.

Now we will look at the anatomy of a function. The function below adds two numbers together and returns the result.

```
unsigned int add(unsigned char A, unsigned char B)
{
    unsigned char answer;
    answer = A + B;
    return answer;
}
```

Here we are defining a function called `add` that takes in two 8 bit numbers, `A` and `B`, adds them together and returns the result, `answer`. The first line of the function starts with a definition of the *type* of the number to be returned, in this case it is `unsigned int`. Data types are discussed more in the Data Types chapter. Then comes the name of the function, `add`, followed by the numbers that are to be read into the function defined in brackets, along with their type - in this case they are both `unsigned char`.

Inside the function itself, we can use the variables `A` and `B` without having to define their type as these have been defined in the function heading. However any other variables, including the one used in the `return` statement, have to be defined in the function itself. In addition, variables `A` and `B` are *local variables* and are limited to use only inside the function, meaning that we can then use different variables, also called `A` and `B`, elsewhere in another function or inside the `main` function without affecting our `add` function. Any variables defined in the function have to be defined at the top of the function, before any lines of code are written.

The function itself is called thus

```
result = add(first_number, second_number);
```

and when implemented in a complete program:

```
void main(void)
{
    unsigned char first_number, second_number;
    unsigned int result;

    first_number = 3;
    second_number = 4;

    result = add(first_number, second_number);
}
```

We notice first that the variables read into our `add` function do not have to have the same variable name, but they must have the same type (`unsigned char`). Therefore it might be more convenient to use variable names `first_number` and `second_number` in the `main` program, but use the variable names `A` and `B` in the function. However, `first_number` must have the same type as `A` and `second_number` must have the same type as `B`. Likewise, the function returns the result, `answer`, and this must have the same type our variable `result`. We also notice that when we call the function, we do not have to include the variable types (eg `unsigned char`) for the input variables.

Now, in C every function has to be *prototyped*. This means we need to tell the compiler that we have defined a function, what it is called with its input and output types. To prototype a function, all we do is copy the heading of the function, but insert a semicolon at the end of the line, thus

```
unsigned int add(unsigned char A, unsigned char B);
```

Ideally, all the function prototypes should be placed before the first function is used and should all be placed together.

If the function does not return a value, then the return type `void` is used. If a function has nothing to return, we can also dispense with the `return` statement at the end of the function. The compiler detects that the end of the function has been reached when it comes to the function's closing bracket. In the same way that some functions do not return variables, other functions might not take any variables as inputs. In this case the keyword `void` is used in place of the input variables in brackets.

## Functions

You will notice that our `main` program used above is in fact a function that takes in no variables and returns no variables so the `void` keyword is used as follows

```
void main(void)
```

It is worth noting that `main` and `interrupt` are special functions and do not need a prototype. Every C program has a `main` function, so the compiler expects it and the `interrupt` function is defined by a keyword `interrupt` and is not called in the traditional way.

We are now going to modify our LED Flasher program to have separate functions to handle the microprocessor initialisation and LED flashing routines.

```
/*-----*/
/* Program Name:   LED Flasher                               */
/*-----*/

#include <htc.h>

__CONFIG(PROTECT & CPD & LVPDIS & BOREN &
          MCLRDIS & PWRTEN & WDTDIS & INTIO);

unsigned char global = 0;

/*-----*/
/* FUNCTION PROTOTYPES                                       */
/*-----*/
void micro_init(void);
void flash_LED (void);

/*-----*/
/* INTERRUPTS                                                */
/*-----*/
void interrupt isr(void) /* int service routine */
{
    if (TMR1IF == 1)
    {
        TMR1IF = 0;
        global = 0x01; /* indicate overflow */
        TMR1H = 0x0B;
        TMR1L = 0xDC;
    }
}

/*-----*/
/* FUNCTION DEFINITIONS                                       */
/*-----*/
/* initialise micro */
void micro_init(void)
```

## Simple C for the PIC Microcontroller

```
{
    /* set ports */
    TRISB = 0b00000001;
    PORTB = 0b00000000;

    /* disable comparator */
    CMCON = 0b00000111;

    /* initialise Timer 1 */
    T1CON = 0b00110100;
    TMR1H = 0x0B;
    TMR1L = 0xDC;

    /* initialise interrupts */
    INTCON = 0b11000000;
    PIE1 = 0b00000001;
    PIR1 = 0b00000000;
}
/*-----*/
/* flash LED */
void flash_LED(void)
{
    /* turn LED ON for 0.5s */
    while((global & 0b00000001)==0)
    {
        PORTB = 0b00000010;
    }
    global = global & 0b11111110;

    /* turn LED OFF for 0.5s */
    while((global & 0b00000001)==0)
    {
        PORTB = 0b00000000;
    }
    global = global & 0b11111110;
}

/*-----*/
/* MAIN PROGRAM */
/*-----*/
void main(void)
{
    micro_init();

    TMR1ON = 1;

    /* flash LED 5 times */
    while(1)
    {
        if(PORTB & 0b00000001)
        {
            flash_LED ();
        }
        else

```

## Functions

```
    {  
        PORTB = 0b00000010;  
    }  
}
```

The code has changed very little from its original form. However, we now have function prototypes defining our two functions

```
void micro_init(void);  
void flash_LED (void);
```

A large comment above these two lines makes the code much easier to read.

Below the interrupt service routine we have defined our two functions. We have done nothing more than a cut and paste exercise on the original code.

It must also be stressed that our two functions are *outside* of the `main` function as are their prototypes. They can be pictured as being a completely separate block of code, independent from the `main` statement that the `main` statement calls upon to perform specific tasks.

In the `main` program we call our two functions using their names. We do not need to include the variable types, and indeed if the function requires no variables as inputs then we can remove the `void` keyword in the function call.

### Variable Scope

We mentioned earlier that variables defined inside a function are only accessible within that function. These are called *local* variables because they can only be used locally within the function. This allows us to use a counter variable, *x*, in two different functions and keep these variables completely separate.

Referring to the program above, if we placed the line

```
unsigned char global = 0;
```

inside the `main` function, we would get a compilation error (undefined variable) in the `interrupt` function as the variable `global` is only defined in the `main` function. Likewise, if we use this line inside the `main` function *and* the `interrupt` function we would be defining two variables – one for

use *only* in the `interrupt` and one for use *only* the `main` function. Our code would compile OK, but would not perform as expected. Our loop

```
while((global & 0b00000001)==0)
{
    PORTB = 0b00000010;
}
```

would inspect bit 0 of the `global` register, but when the interrupt service routine is called, it would modify its copy of the `global` register, but leave the `global` register defined in the `main` function unchanged, so it would never leave the loop.

Global variables need to be defined outside of the functions in which they are going to be used in order to give them global status. Technically they should also be *declared* inside each function that is going to use them using the `extern` `global` keyword thus

```
extern unsigned char global;
```

However, this declaration can be omitted as long as the global variable is declared before it is used and this is indeed what most programmers do. Thus inserting the line

```
unsigned char global = 0;
```

at the topmost part of the code enables us to omit the `extern` declaration.



# Chapter 9

# Working with Binary

# Numbers and Bits

Before we dive into this chapter is worth explaining some terminology. A binary *bit* is single binary digit (either '0' or '1'). Four binary bits make up a nibble (a term that is not often used, but worth knowing anyway) and eight binary bits make up a byte. A byte can represent numbers from 0 to 255 (decimal) and this will be explained later. If we need to represent a number higher than 255 then we need to use more than 8 bits.

We have used binary (base 2), decimal (base 10) and hexadecimal (base 16) numbers in this book so far, but not really explained the benefits of using each. If we are going to program a microcontroller effectively and be familiar with how to manipulate the data read in from the port pins, we need to be conversant with these 3 number bases.

From a very early age, all of us learned to count in base 10. In doing so we count in single digits up to 9 then when the value of 10 is reached, we increment the next 'power of 10' and reset the single digit 'counter' to zero. When we have incremented the digits ( $10^0$ ) as far as they will go, the next digit ( $10^1$ ) is incremented. When this has incremented as far as it will go, the next digit ( $10^2$ ) is incremented and so on.. So the number 1234 can be represented by

$$(1 \times 10^3) + (2 \times 10^2) + (3 \times 10^1) + (4 \times 10^0)$$

The same is true for base 2 (binary). We increment the units counter ( $2^0$ ) from 0 to 1 and when the value of 2 is reached, we increment the next 'power of 2' ( $2^1$ ) and reset the units. Likewise when we have incremented the  $2^1$  digit as far as it will go, we increment the  $2^2$  digit and reset the  $2^1$  digit. Thus the first eight binary numbers are:

000  
001  
010  
011  
100  
101

110

111

We mentioned earlier that an 8 bit binary number (a byte) can represent numbers up to 255 (decimal). Obviously the highest number we can count to with only 8 bits is 11111111 (base 2) and this can be represented as

$$(1 \times 2^7) + (1 \times 2^6) + (1 \times 2^5) + (1 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$$

$$= 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$$

In all counting systems, we increment to the value (base – 1) before incrementing the next ‘power’. So in base 10, we increment to 9 then reset the units and increment the next power ( $10^1$ ). Likewise with base 2, we increment to 1 then reset the units and increment the next power ( $2^1$ )

The same is true for hexadecimal numbers, where we count in base 16. We count from 0 to 15 before incrementing the next power ( $16^1$ ) and resetting the units. However, we run into problems with the hexadecimal counting system in that when we get to the value of *ten* we have not reached the limit of our units – we can still count from ten to fifteen before needing to increment the next power. To get around this, we use the letters A, B, C, D, E, F to represent the numbers ten, eleven, twelve, thirteen, fourteen and fifteen. Thus the hexadecimal counting system is

0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,10,11,12,13,14,15,16,17,18,19,1A,1B,1C,1D,1E,1F,20 etc

So why use binary and hexadecimal when normal decimal will do?

Binary is used extensively in microcontrollers because they operate on 2 logic states (‘0’ and ‘1’), so it is far more convenient to work in numbers related to the power of 2 than those related to the power of 10. It is easy to detect if a voltage is either high or low (5V or 0V). Detecting a voltage in the middle (2.5V) requires a lot more circuitry, so computers have traditionally always operated with just these 2 logic states.

However, working with only 2 digits means that representing a large value means your number can be very long. For example, the value 240 decimal is represented by 11110000 in binary. You can imagine if we have a microcontroller with 1 million different addresses, the corresponding binary number will be huge.

This is where hexadecimal (or *hex*) comes to the rescue. Each 4 binary digits can be represented by 1 hexadecimal digit. This is not surprising since four binary bits enable us to count from 0 to 15 and one hexadecimal digit also allows us to count from 0 to 15. Thus any binary number can be divided into groups of 4 binary digits (starting with the units) and easily turned into the hexadecimal number. Thus

11 0010 0110 1101 1010 1111 0001 in binary

equals

3 2 6 D A F 1 in hex.

It is equally advantageous that larger numbers can be expressed with fewer digits in hex than decimal. The number above occupies only 7 digits in hex, where as its equivalent decimal (52878065) occupies 8.

So when would we use binary over hex numbers and vice versa? This depends on each programmer's preferences, since both numbers are the same. Setting bits in registers and ports is more easily done using binary since each individual bit can be seen. In the following line

```
PORTB = 0b10101010;
```

it is easier to see that every other line has been set to logic 1 than with the line

```
PORTB = 0xAA;
```

When using addresses and variables, binary can soon get out of hand and hex presents a neater solution.

That said, it can get tedious to constantly use `PORTB = 0b11111111` when `PORTB = 0xFF` will do, so this is not a strict rule to live by.

### Logic Functions

Now we have learnt how to switch between binary, decimal and hex bases, it is worth learning how to manipulate binary data.

There are four logical operators in C that can be used to manipulate binary data. These are AND, OR, XOR and NOT and they operate as follows.

The AND (&) operator compares corresponding bits in two binary numbers and gives a resultant bit depending on whether the *both* bits are logic '1' not. Thus

	10101111	binary number 1
&	<u>11110000</u>	binary number 2
=	10100000	result

Here, each bit in the result is '1' if both corresponding bits in binary number 1 and binary number 2 match. The result is '0' if they don't.

The truth table for the AND function is

Inputs	Output
00	0
01	0
10	0
11	1

The OR (|) operator does the same as the AND function, but returns a '1' if *either* of the corresponding bits is set to logic 1, thus

```

    10101010      binary number 1
&   11110000      binary number 2
=   11110100      result
    
```

Here, a comparison of each bit in binary number 1 and binary number 2 shows that only bit 0 and bit 2 have no '1's, thus returning a value of '0' in the result.

The truth table for the OR function is

Inputs	Output
00	0
01	1
10	1
11	1

The XOR (^) operator sets the bits in the result if *only one* bit in either of the binary numbers is set, thus

```

    10101010      binary number 1
&   11110000      binary number 2
=   01011010      result
    
```

We can see that in the above example, only bits 6, 4, 3 and 1 have '1' exclusively in either binary number 1 or binary number 2. If binary number 1 or binary number 2 have the same corresponding bits then the result is '0'.

The truth table for the XOR function is

Inputs	Output
00	0
01	1
10	1
11	0

The NOT (~) operator simply inverts all the bits of the binary number and hence only requires one binary number to act on. Thus

$$\sim(11110000) = 00001111$$

So why is this useful? Designing a digital system that reads values in from port pins means that sooner or later we are going to have to manipulate those values. Data read in from an 8 way data port is nothing more than an 8 bit binary number.

We saw in our first program the line

```
if(PORTB & 0b00000001)
```

This is a test to see if bit 0 in Port B is at a high voltage (logic 1) or a low voltage (logic 0).

We mentioned that the `if` statement executes if the evaluation takes on a value of '1' (true) or '0' (false) but more precisely it executes if the evaluation is *non-zero*. Thus we could simply write

```
if(PORTB)
{
    /* code goes here */
}
```

and if Port B bit 0 is set, we would effectively be evaluating

```
if(00000001)
{
    /* code goes here */
}
```

and our `if` statement would execute. If Port B bit 0 were cleared we would effectively be evaluating

```
if(00000000)
{
    /* code goes here */
}
```

and our `if` statement would not execute. However, this leaves us vulnerable to other bits in Port B being set. If bit 7 were set, then our `if` statement would always execute regardless of the state of bit 0 (since the value of Port B would always be non-zero).

Thus we need a way of masking out the other bits in Port B and only looking at bit 0. This is an ideal job for the AND function and by ANDing the value read in from Port B, we can force certain bits to '0'.

From the following example

	10101010	data read in from Port B
&	<u>11110000</u>	Data that we AND with Port B
=	10100000	result

we can see that regardless of the state of the bits on Port B, we can force them to '0' just by ANDing them with a corresponding '0', as shown in red above. (note that we are not changing the port value, just the result read in from the port). If we AND them with a '1' the data remains unchanged, as shown in blue above.

Thus in the following example

	10101010	data read in from Port B
&	<u>00000010</u>	Data that we AND with Port B
=	00000010	result

we can inspect *only* bit 1 and see if it is a logic '1' or '0', with the result evaluating to zero if bit 1 is '0' and *non-zero* if bit 1 is '1'.

## Working with Binary Numbers and Bits

While the AND function is useful for clearing bits, the OR function is good for setting bits. Suppose we need to make sure a bit in a register is definitely set. We can see from the example below

	10101010	register value
&	11110000	binary number we OR with register value
=	11111010	new register value

that ORing our register value with 11110000 ensures that the new register value always has the top four bits set regardless of their original value, as shown in red. ORing the register bits with '0' keeps the bits unchanged, as shown in blue.

Suppose we need to change the state of a bit from '1' to '0' or from '0' to '1'. Instinctively the NOT function comes to mind. However, the NOT function only works on the entire register and changes the state of all the bits. This is where the XOR function can be used.

From our earlier example:

	10101010	binary number 1
&	11110000	binary number 2
=	01011010	result

we can see that the top 4 bits in binary number 1, when XORed with the corresponding logic '1' in binary number 2 actually change state, while the bits XORed with logic '0' remain unchanged. So if an individual bit inside a register needs to be changed, XORing that bit with a logic '1', keeping all the other bits at logic '0' will change only that bit.

With this in mind, our LED flasher function can now be considerably reduced. Below is our new code:

```
/*-----*/
/* flash LED */
void flash_LED (void)
{
    /* turn LED ON for 0.5s */
    while((global & 0b00000001)==0)
    {
        ;
    }
    global = global & 0b11111110;
    PORTB = PORTB ^ 0b00000010;
}
/*-----*/
```

We have replaced 2 **while** statements with just one. Instead of setting then clearing bit 1 in Port B using 2 **while** statements, we are now using the XOR operator (^) to toggle Port B bit 1 between '0' and '1'. Note also the use of the & function to ensure that `global` bit 0 is cleared.

Finally, we have also used the Null Statement (;). This is a statement that does nothing, but just stops our compiler getting upset if it sees no instructions inside the **while** loop. This is not often seen in C, but is a perfectly valid command.

### Shifting Bits

There are two more operators that are often used on binary numbers. These are the << and >> operators.

The operator << shifts the binary byte a specified number of places to the left. This has the same effect as doubling the binary number. If you consider that shifting a decimal number one place to the left has the same effect as multiplying the number by 10, then it is easy to see that shifting a binary number one place to the left multiplies the number by 2.

The operator >> shifts the binary byte a specified number of places to the right, halving the number.

The two operators above have all sorts of useful applications, but are most commonly used when setting, clearing and checking bits.

To set bit 4 in a register the following line is often used:

```
register = register |(1<<4);
```

Here we are shifting a single logic '1' left by 4 positions and ORing `register` with the result. This has the effect of setting bit 4 in `register`. Naturally, the number 4 can be replaced with any number you like from 0 to the size of the register.

Likewise if we need to clear a bit 4 in a register the following line is often used

```
register = register & ~(1<<4);
```

Again we are shifting a single logic '1' left by 4 positions, inverting all the bits (using the NOT function) then ANDing this number with `register`.

Thus 00000001 is shifted 4 places to the left to give 00010000. This is inverted to give 11101111. This is ANDed with `register` to clear bit 4 in `register`.



Finally, if we need to check if bit 4 is '1' or '0', we can use the following

```
register & (1<<4);
```

This shifts a single logic '1' left by 4 places and ANDs the result with `register`. If bit 4 is '1' then the result returned is non zero. If bit 4 is cleared, then the result is zero.

Now let's write a program demonstrating some of the above techniques.

This program uses our existing LED flasher, but in addition has a Knight Rider strobe (LEDs are lit in sequence: 1,2,3,4,5,6,7,8,7,6,5,4,3,2,1) as well as an 'alternate flash' where LEDs 1,3,5,7 flash on and off. We have also revised some of our earlier code to neaten it up a bit.

The Simple Flash routine (from before) is shown below, but this time flashes 10 times without the intervention of a button press. Note that one 'ON' and one 'OFF' is counted as 2 cycles in our count below

```
/* flash LED */
void simple_flash(void)
{
    unsigned char n;

    for(n=0; n<20; n++)
    {
        /* turn LED ON for 0.5s */
        while((global & 0b00000001)==0)
        {
            ;
        }
        global &= 0b11111110;
        PORTB ^= 0b00000001;
    }
}
```

The Knight Rider strobe consists simply of loading in a binary number using the line

```
PORTB = 0x01; /* far right LED ON */
```

then using the command << to shift the data left consecutively until a count of 7 is reached, then shifting the data back again, displaying the data on the port pins as we go. The program below is an example of this

```
/* knight rider */
void knight_rider(void)
{
    unsigned char n, count;
    count = 0;
    PORTB = 0x01;                                /* far right LED ON */

    /* strobe LEDs back and forth */
    for(n=0; n<10; n++)
    {
        for(count=0; count<7; count++)
        {
            while((global & 0b00000001)==0)
            {
                ;                                /* 0.5 seconds delay */
            }
            global &= 0b11111110;
            PORTB <<= 1;
        }
        for(count=0; count<7; count++)
        {
            while((global & 0b00000001)==0)
            {
                ;                                /* 0.5 seconds delay */
            }
            global &= 0b11111110;
            PORTB >>= 1;
        }
    }
}
```

Our Knight Rider strobe uses the lines

```
global &= 0b11111110;
PORTB >>= 1;
```

which is a shorter way of writing

```
global = global & 0b11111110;
PORTB = PORTB >> 1;
```

Shortcuts are found in virtually all C programs and present a neater way of programming. These are explained in more detail in Appendix B.

The alternate flash routine makes use of the XOR function which is good for flipping the state of bits. Again, one 'ON' and one 'OFF' is counted as 2 cycles:

```
/* alternate LED flash */
void alt_flash(void)
{
    unsigned char n;
    PORTB = 0x00;

    for(n=0; n<20; n++)
    {
        /* turn LED ON for 0.5s */
        while((global & 0b00000001)==0)
        {
            ;
        }
        global &= 0b11111110;
        PORTB ^= 0b10101010;
    }
}
```

The complete code is shown below

```
/*-----*/
/* Program Name:    LED Flasher                               */
/*-----*/

#include <htc.h>

__CONFIG(PROTECT & CPD & LVPDIS & BOREN &
          MCLRDIS & PWRTEN & WDTDIS & INTIO);

unsigned char global = 0;

/*-----*/
/* FUNCTION PROTOTYPES                                       */
/*-----*/
void micro_init(void);
void simple_flash(void);
void knight_rider(void);
void alt_flash(void);

/*-----*/
/* INTERRUPTS                                                */
/*-----*/
```

## Simple C for the PIC Microcontroller

```
/*-----*/
void interrupt isr(void)                               /* int service routine */
{
    if (TMR1IF == 1)
    {
        TMR1IF = 0;
        global = 0x01;                               /* indicate overflow */
        TMR1H = 0x0B;
        TMR1L = 0xDC;
    }
}
/*-----*/
/* FUNCTION DEFINITIONS                               */
/*-----*/
/* initialise micro */
void micro_init(void)
{
    /* set ports */
    TRISB = 0x00;                                     /* port B is all outputs */
    PORTB = 0x00;

    /* disable comparator */
    CMCON = 0b00000111;

    /* initialise Timer 1 */
    T1CON = 0b00110100;
    TMR1H = 0x0B;
    TMR1L = 0xDC;

    /* initialise interrupts */
    INTCON = 0b11000000;
    PIE1 = 0b00000001;
    PIR1 = 0b00000000;
}
/*-----*/
/* flash LED */
void simple_flash(void)
{
    unsigned char n;

    for(n=0; n<20; n++)
    {
        /* turn LED ON for 0.5s */
        while((global & 0b00000001)==0)
        {
            ;
        }
        global &= 0b11111110;
        PORTB ^= 0b00000001;
    }
}
/*-----*/
/* knight rider */
void knight_rider(void)
```

## Working with Binary Numbers and Bits

```
{
    unsigned char n, count;
    count = 0;
    PORTB = 0x01;                                     /* far right LED ON */

    /* strobe LEDs back and forth */
    for(n=0; n<10; n++)
    {
        for(count=0; count<7; count++)
        {
            while((global & 0b00000001)==0)
            {
                ;                                     /* 0.5 seconds delay */
            }
            global &= 0b11111110;
            PORTB <<= 1;
        }
        for(count=0; count<7; count++)
        {
            while((global & 0b00000001)==0)
            {
                ;                                     /* 0.5 seconds delay */
            }
            global &= 0b11111110;
            PORTB >>= 1;
        }
    }
}
/*-----*/
/* alternate LED flash */
void alt_flash(void)
{
    unsigned char n;
    PORTB = 0x00;

    for(n=0; n<20; n++)
    {
        /* turn LED ON for 0.5s */
        while((global & 0b00000001)==0)
        {
            ;
        }
        global &= 0b11111110;
        PORTB ^= 0b10101010;
    }
}
/*-----*/
/* MAIN PROGRAM */
/*-----*/
void main(void)
{
    micro_init();
    TMR1ON = 1;
}
```

## Simple C for the PIC Microcontroller

```
while(1)
{
    simple_flash();
    knight_rider();
    alt_flash();
}
```

# Chapter 10

# Data Types

Before we use any variables in C, we have to tell the compiler what data type they are. This informs the compiler how many bits to assign to storing the variable. Generally once a variable has been declared as a certain data type, it remains as that data type throughout the entire program.

The PIC only has a limited amount of RAM (the PIC16F627A has 224 bytes), unlike most PCs that have several trillion bytes. Therefore we have to be very conservative with the way we use it. Because most of the interface with the outside world is done via the 8 bit port pins of the PIC and most of the internal registers are 8 bits wide most of the data manipulation inside the PIC can be kept to 8 bits. The table below shows the data types that are available.

Type	Size (bits)	Arithmetic Type
<code>bit</code>	1	unsigned integer
<code>char</code>	8	signed or unsigned integer
<code>unsigned char</code>	8	unsigned integer
<code>short</code>	16	signed integer
<code>unsigned short</code>	16	unsigned integer
<code>int</code>	16	signed integer
<code>unsigned int</code>	16	unsigned integer
<code>short long</code>	24	signed integer
<code>unsigned short long</code>	24	unsigned integer
<code>long</code>	32	signed integer
<code>unsigned long</code>	32	unsigned integer
<code>float</code>	24	real
<code>double</code>	24 or 32	real

We note that an 8 bit variable can be specified as either `char` or `unsigned char`. `char` is a *signed* data type meaning it enables us to represent both positive and negative numbers. The price we pay for this is that we lose the most significant bit (MSB, or bit 7) as this is now used as a *sign* bit. Therefore with the data type `char` we can count from +127 to -128. Since most of the time we only deal with positive numbers (and indeed all of our programs so far have only handled positive numbers), having a sign bit is of little use, so we elect to use the `unsigned char` data type allowing us to count from 0 to 255. Don't think that the `char` data type is only applicable to characters

though. Any variable that can take on a value from 0 to 255 can be assigned the `unsigned char` data type.

Now supposing we need to use a variable that might contain a value higher than 255. Then we must move up to the next data type, `unsigned int`. The Hi Tech compiler allows us to also use the data type `unsigned short` but `unsigned int` is the more popular term to use for 16 bit data variables and allows us to represent values from 0 to 65535. Again if we omit the `unsigned` keyword, then the compiler assumes we are referring to a signed variable, so we are only allowed to use numbers in the range +32767 to -32768. Since most of our work with the PIC deals with positive numbers, the `unsigned int` data type is the preferred one.

We have already come across the `unsigned int` data type before in our Functions chapter in the function shown below.

```
unsigned int add(unsigned char A, unsigned char B)
{
    unsigned char answer;
    answer = A + B;
    return answer;
}
```

Variables *A* and *B* can take on a value of up to 255, but if added together can result in a number greater than 255, so we had to define the `answer` variable as `unsigned int`.

All of the above also applies to data types `unsigned short long` and `unsigned long` allowing us variables up to 24 bits and 32 bits respectively. However these data types are very rarely used with the PIC microcontroller.

### Converting Data Types

We have explained previously that once a variable is assigned a type, it cannot be changed. This is not strictly true and C allows us to ‘coerce’ a variable into a different type. With the PIC, an example of this would be where we read in an 8 bit variable from the port pins, but we need to perform 16 bit manipulation on that data. If the port pins represent the lower 8 bits of a 16 bit `unsigned int` variable, the 8 bit `unsigned char` variable from our port pins will have to be coerced into a variable of type `unsigned int`. This is done as follows

```
unsigned char port_var;
unsigned int int_var;

int_var = 0;
port_var = PORTB;
int_var = (int)port_var;
```



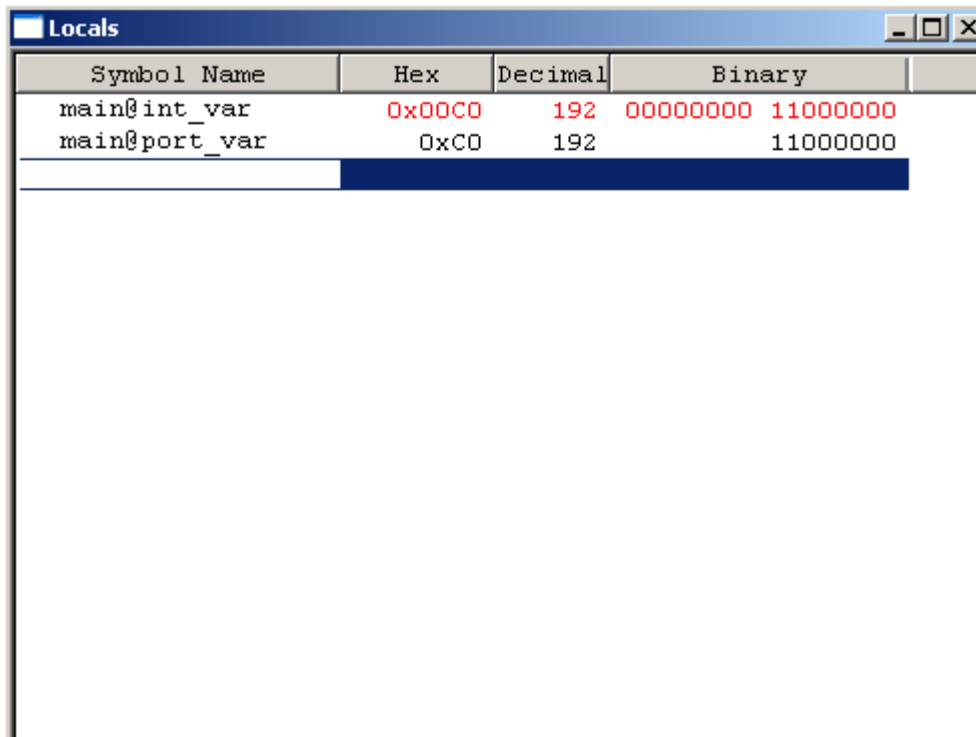
## Data Types

We see that a 16 bit variable, `int_var`, as well as an 8 bit variable, `port_var` have been declared. We read in the value of the Port B register into `port_var`, but need to perform 16 bit manipulation on it. The line

```
int_var = (int)port_var;
```

coerces `port_var` into a 16 bit variable.

MPLAB shows the result of the above code in the Locals window where we can clearly see the 16 bit variable `int_var` containing the data from the variable `port_var`:



The screenshot shows the 'Locals' window in MPLAB. It contains a table with the following data:

Symbol Name	Hex	Decimal	Binary
main@int_var	0x00C0	192	00000000 11000000
main@port_var	0xC0	192	11000000

# Chapter 11

## Arrays

So far we have worked with LEDs and indeed the code we have used to flash the LEDs can be found in millions of C programs across the world. We are now going to investigate LCDs (liquid crystal displays) and in doing so learn about another aspect of C: Arrays. A typical LCD datasheet is available from the SimonBramble Website and it might be prudent to download this to help with the understanding of LCDs and their addressing modes.

Arrays are a convenient way of storing lots of (often related) variables all with the same data type and we are going to write a small LCD routine to demonstrate this. The LCD display we are going to use consists of 2 rows, each row with 16 characters as typically used in swipe card machines, faxes, photocopiers and many consumer electronics items. Each character has a unique code (an ASCII code – see Appendix G) that determines exactly what letter, number or symbol it is and whether it is upper or lower case. ASCII codes span from 0 to 255 so can be represented as variables with the `unsigned char` data type. Rather than assign 32 individual variable names to each character (then trying to remember which variable is assigned to which LCD character), we are going to put all of our variables into two arrays – one for row 1 of our LCD and one for row 2.

There are a number of different ways of declaring an array. The following line declares an array, called `row1` of `unsigned char` that is 16 elements long.

```
unsigned char row1[16];
```

The array name is `row1` and has to follow all the same guidelines for naming other variables as outlined in Appendix A. All elements inside the array have to have the same data type, in this case `unsigned char`. The number in the square brackets tells us how many elements the array can hold, in this case 16 elements, each element 8 bits in length. If the array was declared to consist of `unsigned int` then the compiler would assign storage space for 16 elements, each 16 bits in length.

The array elements are labelled 0 to 15 with the first element being element 0 and the last element 15, thus addressing element 1 means we are inspecting the *second* element in the array. This is most important and easily forgotten.

We now need to fill the array. An example of how to assign a value to a single array element is shown below.

## Arrays

```
row1[0] = 10;
```

This assigns the value 10 to the first element of the array `row1`.

Like any other variable, arrays can be inserted into loops and filled, thus

```
unsigned char row1[16];
unsigned char n;

for(n=0; n<16; n++)
{
    row1[n] = n;
}
```

This function fills the elements of `row1` with the numbers from 0 to 15. Note that the `for` loop counts from 0 to 15. It would be incorrect to use the following line

```
for(n=1; n<17; n++)
```

as this would leave `row1[0]` unfilled, but more importantly try to fill element 16 which does not exist. It is interesting to note that the above code does in fact compile correctly. Leaving `row1[0]` unfilled (or uninitialized) is perfectly legal, as long as you do not rely on it holding non-random data further on in your program.

Another way of filling the array is as follows, at declaration stage

```
unsigned char row1[16] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
```

or we can omit the array dimension and let the compiler decide how much space to assign given the initialisation values thus

```
unsigned char row1[] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
```

Alternatively we can declare the array in one step, then initialise it in another, thus

```
unsigned char row1[16];  
unsigned char row1[16] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
```

It is advisable to stick with only one of the above solutions. Leaving the brackets empty and letting the compiler decide how big the array should be is the preferred method for single dimensional arrays.

Multidimensional arrays can also be declared by specifying first the *row size* then *column size* of the array thus

```
unsigned char row[2][16] = {  
0 ,1 ,2 ,3 ,4 ,5 ,6 ,7 ,8 ,9 ,10,11,12,13,14,15,  
16,17,18,19,20,21,22,23,24,25,26,26,27,28,30,31  
};
```

Here we make use of rigid formatting of the text that the compiler ignores, but makes the code much more readable. We could have declared the array all on one long line, but it would not be obvious to the reader that we are defining 2 rows, each of 16 columns. Inserting spaces where appropriate makes this a lot more visible. Since multidimensional arrays are a bit more confusing, you need to specify the rows and columns at declaration stage (you cannot leave the square brackets empty). Initialising them at declaration stage also keeps things simple.

In truth any number of array dimensions can be specified, although more than 3 dimensions starts to get hard to visualise. Here is a 3 dimensional array consisting of 3 rows of 4 columns, and 2 layers deep. The format is

```
unsigned char cube[layer][row][column]
```

and a practical example is:

```
unsigned char cube[2][3][4] = {  
1 ,2 ,3 ,4 ,  
5 ,6 ,7 ,8 ,  
9 ,10,11,12,  
  
13,14,15,16 ,  
17,18,19,20 ,  
21,22,23,24  
};
```

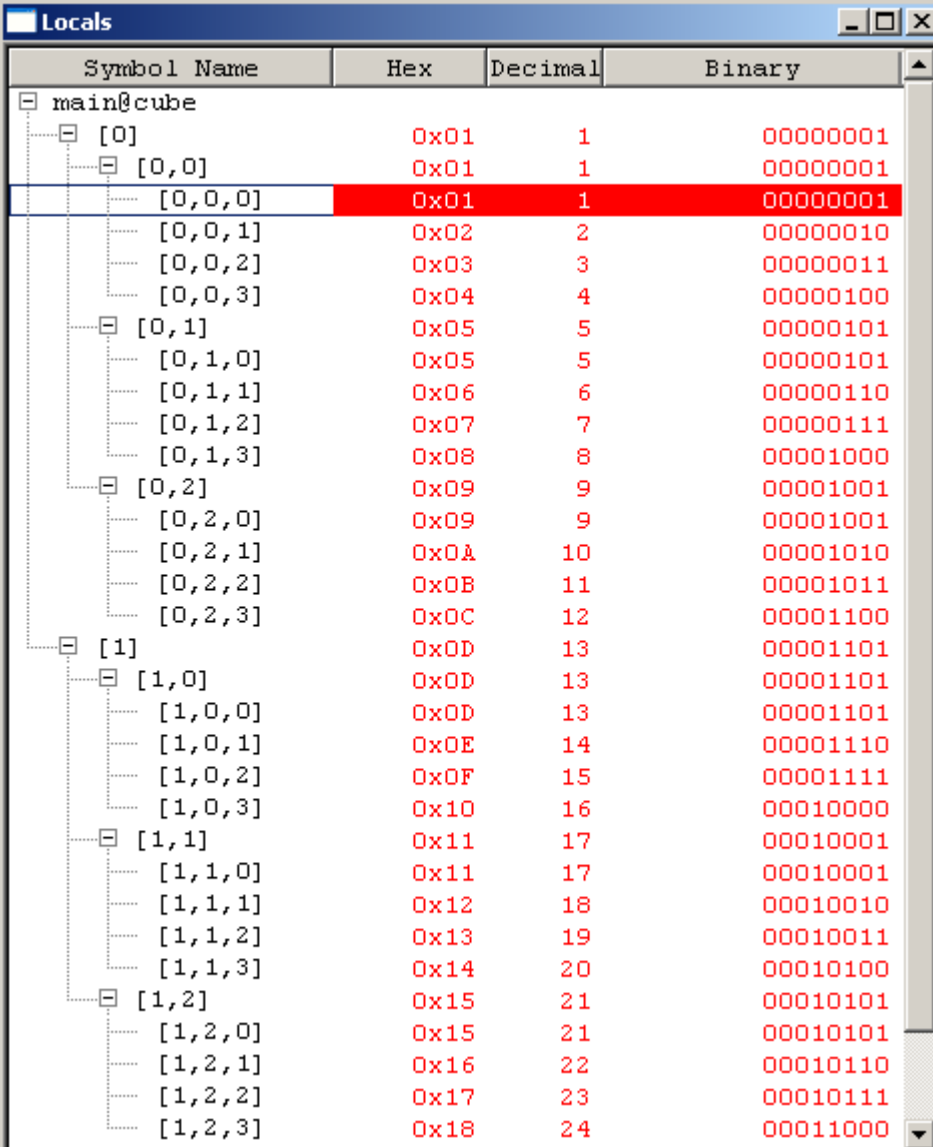
## Arrays

Remembering that all arrays start with element 0, we can say that

```
cube[1][2][3] = 24
```

since the first 'block' of numbers is `cube[0][x][x]`, so we are picking an element from the second block of numbers, 3<sup>rd</sup> row, 4<sup>th</sup> column.

If you simulate this in MPLAB, the Locals window gives a handy display of a 3 dimensional array:



Symbol Name	Hex	Decimal	Binary
main@cube			
[0]	0x01	1	00000001
[0,0]	0x01	1	00000001
[0,0,0]	0x01	1	00000001
[0,0,1]	0x02	2	00000010
[0,0,2]	0x03	3	00000011
[0,0,3]	0x04	4	00000100
[0,1]	0x05	5	00000101
[0,1,0]	0x05	5	00000101
[0,1,1]	0x06	6	00000110
[0,1,2]	0x07	7	00000111
[0,1,3]	0x08	8	00001000
[0,2]	0x09	9	00001001
[0,2,0]	0x09	9	00001001
[0,2,1]	0x0A	10	00001010
[0,2,2]	0x0B	11	00001011
[0,2,3]	0x0C	12	00001100
[1]	0x0D	13	00001101
[1,0]	0x0D	13	00001101
[1,0,0]	0x0D	13	00001101
[1,0,1]	0x0E	14	00001110
[1,0,2]	0x0F	15	00001111
[1,0,3]	0x10	16	00010000
[1,1]	0x11	17	00010001
[1,1,0]	0x11	17	00010001
[1,1,1]	0x12	18	00010010
[1,1,2]	0x13	19	00010011
[1,1,3]	0x14	20	00010100
[1,2]	0x15	21	00010101
[1,2,0]	0x15	21	00010101
[1,2,1]	0x16	22	00010110
[1,2,2]	0x17	23	00010111
[1,2,3]	0x18	24	00011000

If you are totally crazy, C allows you to specify arrays with more than 3 dimensions, but most programs can be adequately achieved with only 2 dimensions.

Text is defined slightly differently in arrays. In a text array the last character has to be a *null character*. This is represented by the symbol `\0`. This means that if we declare an array, it can hold one less character than we specify since the last character has to be a `\0`. Thus

```
unsigned char row1[14] = {  
'E', 'L', 'E', 'C', 'T', 'R', 'O', 'N', 'W', 'O', 'R', 'K', 'S', '\0'  
};
```

*It is interesting to note that if we reduce the dimension of `row1` from 14 to 13 elements and leave off the `\0` on the end of our character string, the code still compiles in MPLAB and simulates OK, implying that MPLAB is happy with character strings unterminated with the null character. However, reducing the dimension to 12 does cause a 'too many initializers' error.*

As with single dimension arrays, we can let the compiler decide how much space we need if we use the following format. Indeed this is the more common format:

```
unsigned char row1[] = "ELECTRONWORKS";
```

Using this format, the compiler also automatically inserts the `\0` character at the end of our string. By putting the character string in quotes also instruct the compiler to store the ASCII code of each letter in the array elements.

We are now going to write a program to display on our LCD:

```
ELECTRONWORKS  
Simple C for PIC
```

The most common alphanumeric LCDs are driven by a Hitachi HD44780 driver. The datasheet is readily downloadable, but is not easy to understand so leaves many engineers pondering why their LCD display does not work. We have written a bullet proof LCD program that has behaved well for years and this will be used here.

The HD44780 generates the following characters:

## Arrays

Lower 4 Bits	Upper 4 Bits	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
xxxx0000	CG RAM (1)			0	a	P	`	P				-	夕	ミ	α	ρ	
xxxx0001	(2)		!	1	A	Q	a	q			。	ア	チ	△	ä	q	
xxxx0010	(3)		"	2	B	R	b	r			「	イ	ツ	×	ρ	θ	
xxxx0011	(4)		#	3	C	S	c	s			」	ウ	テ	モ	ε	ε	
xxxx0100	(5)		\$	4	D	T	d	t			、	エ	ト	ト	μ	Ω	
xxxx0101	(6)		%	5	E	U	e	u			・	オ	ナ	工	α	Ü	
xxxx0110	(7)		&	6	F	V	f	v			ヲ	カ	ニ	ヨ	ρ	Σ	
xxxx0111	(8)		'	7	G	W	g	w			ア	キ	ヌ	ラ	g	π	
xxxx1000	(1)		(	8	H	X	h	x			イ	ク	ネ	リ	γ	×	
xxxx1001	(2)		)	9	I	Y	i	y			ウ	ケ	ル	ル	γ	γ	
xxxx1010	(3)		*	:	J	Z	j	z			エ	コ	ハ	レ	j	キ	
xxxx1011	(4)		+	:	K	[	k	(			オ	サ	ヒ	ロ	*	π	
xxxx1100	(5)		,	<	L	¥	l	l			カ	シ	フ	フ	φ	π	
xxxx1101	(6)		-	=	M	]	m	)			ユ	ヌ	ハ	ン	モ	÷	
xxxx1110	(7)		.	>	N	^	n	÷			ヨ	セ	ホ	°	ñ		
xxxx1111	(8)		/	?	O	_	o	€			ッ	ソ	マ	°	ö		

Note in the top left corner of the table, it states that the binary numbers along the top from 0000 to 1111 represent the upper 4 bits of the character code and the binary numbers in the left hand column represent the lower 4 bits of the character code. This character table is loosely based on the ASCII

character set as outlined in Appendix G. We saw above that reading text into an array using the format

```
unsigned char row1[] = "ELECTRONWORKS";
```

automatically converts each character into its equivalent ASCII code and places it into our array, therefore we can read our ASCII values directly out of the array into the LCD controller without any complex maths in between.

Now, most 2 line LCD displays based on the HD44780 controller have 8 data pins, 3 supply pins and 3 control pins as described below.

Pin No.	Symbol	Function
1	Vss	Ground
2	Vdd	Supply 2.7V to 5.5V
3	Vo	Liquid Crystal Supply
4	RS	Register Select
5	R/W	Read/Write
6	E	Enable
7	DB0	DB0
8	DB1	DB1
9	DB2	DB2
10	DB3	DB3
11	DB4	DB4
12	DB5	DB5
13	DB6	DB6
14	DB7	DB7
15	N/A	Backlight
16	N/A	Backlight

Although it provides 8 data pins, the HD44780 also allows us to interface it to a microprocessor using only the top 4 data lines (DB4 – DB7), thus saving us 4 port pins. This makes the code slightly more complicated, but it is worth it to save on port pins. The code outlined here works with a 4 bit wide data bus.

Put simply, either commands or data can be written to the LCD display. Commands instruct the LCD to clear the display, move the cursor etc. whereas data tells the LCD what characters to display. If the Register Select pin (RS) is held low, the LCD interprets the data on pins DB4 – DB7 as a command whereas if the RS pin is held high, the data on pins DB4 – DB7 is interpreted as character information.

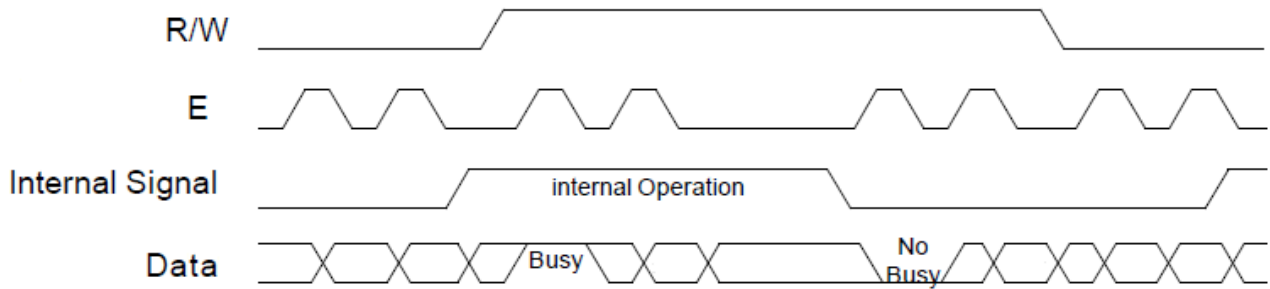


## Arrays

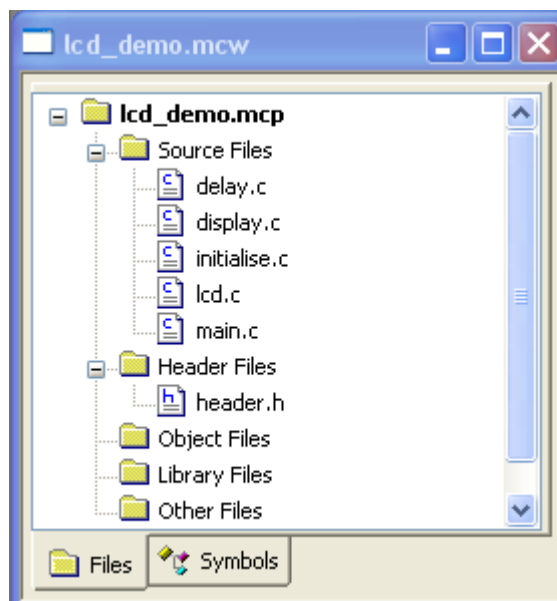
The Enable pin (E) clocks in the data. The enable pin is set high, the data is presented on pins DB4 – DB7, then the enable pin is set low, clocking in the data.

The R/W pin determines if we are reading to or writing from the LCD. Since we will always be writing to the LCD, we will tie this pin permanently low and not connect it to a port pin.

The write sequence is shown graphically in the figure below



The code for implementing this is shown later. The code is written in a series of files with each file containing one or more functions of our program. Each file was written then saved, then added to the MPLAB by right clicking over the Source Files heading in the Project Window of `lcd_demo.mcw`.



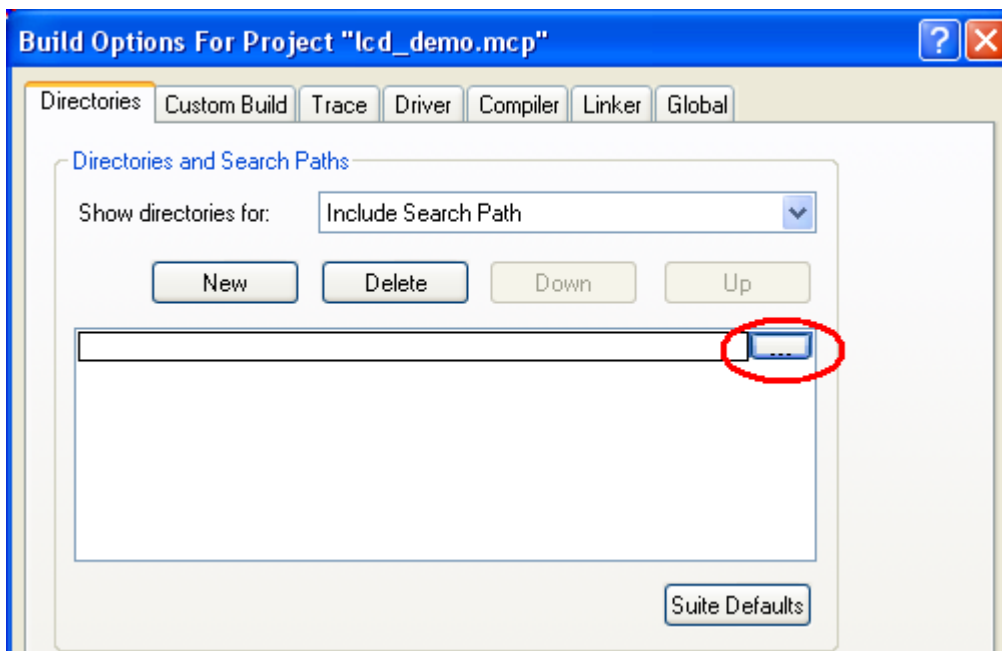
These functions are called by `main.c`.

The files that make up the complete program are: `main.c`, `delay.c`, `display.c`, `initialise.c`, `lcd.c` and a header file, `header.h`.

## Simple C for the PIC Microcontroller

As a general rule, all port pin definitions, function prototypes and variables common to all program files should be placed in the header file. Picture the header file as being a file that stores all the 'junk' common to all the other program files. The actual functions themselves are stored in separate '.c' files.

If you are going to write code spread over several files, MPLAB needs to be told where to look for them when it compiles them. This is done by going to the main menu and selecting Project -> Build Options -> Project. Then in 'Directories and Search Paths' click the dropdown menu for 'Show Directories For' and select 'Include Search Path'. Click 'New' then click the Directories button:



Then select the directory in which your project is based and click OK. Thus MPLAB will search this directory for all your files when it comes to compiling your project.

Here are the final files:

main.c File:

Firstly, we have declared 2 arrays, `string1[]` and `string2[]` to hold the characters for row 1 and row 2 respectively.

In `main.c` we call functions `micro_initialise` and `lcd_initialise` that are both stored in file `initialise.c`. `lcdwrite` is a function that presents the data to the microprocessor port pins and is stored in the file `lcd.c`. `showrow` is a function stored in `display.c` and reads the characters out of our array and puts them on the LCD display.

## Arrays

```
/* ===== */
/* Program Name: LCD Program */
/* Software By: Simon Bramble */
/* Date: March 2010 */
/* Processor: PIC 16F627A */
/* Clock Speed: Internal 4MHz oscillator */
/* Compiler: HI-TECH/MPLAB compiler */
/* ===== */

/* ===== */
/* INCLUDES */
/* ===== */
#include <htc.h>
#include <header.h>

__CONFIG(PROTECT & CPD & LVPDIS & BOREN &
          MCLRDIS & PWRTEN & WDTDIS & INTIO);

/* ===== */
/* INTERRUPTS */
/* ===== */
void interrupt isr(void) /* int service routine */
{
    setbit(global, 0);
    TMR1IF = 0;
}
/* ===== */
/* MAIN PROGRAM */
/* ===== */
void main(void)
{
    unsigned char string1[] = " ELECTRONWORKS ";
    unsigned char string2[] = "Simple C for PIC";

    micro_initialise(); /* initialise micro */
    lcd_initialise(); /* Initialise LCD */

    // set row 1
    lcd_write(0b10000000); /* move cursor to start line 1 */
    show_row(string1);
    // set row 2
    lcd_write(0b11000000); /* move cursor to start line 2 */
    show_row(string2);

    while(1)
    {
        ;
    }
}
```

### Header.h file

You will notice we have also added a header.h file to help de-clutter the program. Inside the header file we have put all the port pin definitions and the function prototypes. We have also defined some *macros*. Macros are explained in more detail in Appendix C. These allow us to replace code with statements that are more meaningful and easier to read. We have defined 2 macros

```
#define setbit(address, bit) (address |= (1<<bit))
#define clearbit(address, bit) (address &= ~(1<<bit))
```

At compile stage, the compiler replaces the text

```
setbit(address, bit)
```

with

```
(address |= (1<<bit))
```

The above code allows us to write for example

```
setbit(PORTA, 0)
```

to set Port A bit 0 instead of having to write

```
PORTA |= 0
```

It makes the code a lot easier to read. Note that macros are defined with the **#define** preprocessor directive and do not have a semicolon at the end of the line (like **#include** statements), but when used in code are treated as a normal statement and must have a semicolon included at the end of the line.

We have also defined our global variable `global` to act as a general purpose status register. In the code, we set bit 0 if Timer 1 has overflowed and we use bit 1 to determine if we are writing to the LCD using one 4-bit transfer or two – see later.

## Arrays

If we are going to use the macros defined in `header.h` in any other of our files then we have to include `header.h` in each of those files using the line

```
#include <header.h>
```

This leads to the possibility that the compiler could get upset if it sees more than one definition of a variable defined in the header file. In other words, if we define a variable (for example `unsigned char global;`), then include the header file in two or more other files (using `#include <header.h>`), the compiler might see this as being a multiple definition of the variable `global`.

Therefore, we use the preprocessor directives:

```
#ifndef HEADER_H_
#define HEADER_H_

/* header contents go here*/

#endif /*HEADER_H_*/
```

at the top and bottom of our header file. The statement `#ifndef` is like the `if` statement, but is an instruction to the compiler. It evaluates if the header file has already been defined and if it has not, it defines it. Thus any variable defined in our header file will only be compiled once.

Here is the `header.h` file:

```
#ifndef HEADER_H_
#define HEADER_H_

/*-----*/
/* GLOBALS */
/*-----*/
unsigned char global; /* gen purpose register */
/* global bits:
   b0 = has timer overflow occurred?
   b1 = are we doing 1 or 2 4 bit transfers?
*/
/*-----*/
/* PIN DEFINITIONS */
/*-----*/
/* PORT A PINS */
#define E 0 /* enable */
#define RS 1 /* data/command mode */
```

## Simple C for the PIC Microcontroller

```
/* PORT B PINS */
/* RB7:RB4 = data lines DB7:DB4 */

/*-----*/
/* MACROS */
/*-----*/
#define      setbit(address, bit)      (address |= (1<<bit))
#define      clearbit(address, bit)    (address &= ~(1<<bit))

/* FUNCTION PROTOTYPES =====*/
void interrupt isr(void);              /* int service routine */
void micro_initialise(void);           /* Initialise Micro */
void lcd_initialise(void);             /* Initialise LCD */
void lcd_write(unsigned char data);    /* 4 bit LCD transfer */
void load_lcd(unsigned char data);     /* load data into LCD */
void show_row(unsigned char string[]); /* display row on lcd */
void delay_10ms(void);                /* 10ms delay */

#endif /*HEADER_H_*/
```

### Initialise.c File

The initialisation of our microprocessor and LCD is done inside `initialise.c`. We have discussed the initialisation of the microprocessor and this has not changed from before.

The initialisation of the LCD is the tricky bit. Since we are starting with commands (not data) the RS pin is held low. We start with a delay routine of 100ms to allow the LCD to power up and for the power supply rails to settle. This is followed by 3 cycles of writing 0011 to DB7 – DB4 (clocking the E pin as we go). We then need to tell the LCD that the interface is 4 bits wide and this is done by writing 0010 to DB7 – DB4 twice. In the second parse, we also instruct the LCD to use 2 lines and 5x7 matrix font using the code 0010 1000. We then switch the display off, cursor off and blink off using the byte 0000 1000. We then clear the display with the byte 0000 0001 and finalise the initialisation with the byte 0000 0110. Various delays are meant to be inserted between each bus transaction, but in our code, we simply use a delay of 10ms.

The function `lcd_initialise()` is fairly straight forward. We have a delay of 100ms at the start to wait for the power supply to settle. We then write data to the LCD using either one or two 4-bit data transfers. The function `lcd_write()` (stored in the `lcd.c` file) uses the status of bit 1 in the `global` register to determine if the transfer is a one or two 4-bit transfer. A typical LCD datasheet is available from the SimonBramble website and it might be prudent to download this to help with the understanding of LCDs and their addressing modes.

Here is the `initialise.c` file:

## Arrays

```
#include <header.h>
#include <htc.h>
/* ===== */
/* INITIALISE MICRO */
/* ===== */
void micro_initialise(void)
{
    /* set ports */
    TRISA = 0x00;          /* port A is all outputs */
    PORTA = 0x00;
    TRISB = 0x00;          /* port B is all outputs */
    PORTB = 0x00;

    /* disable comparator */
    CMCON = 0b00000111;

    /* initialise Timer 1 */
    T1CON = 0b00110100;
    TMR1H = 0x0B;
    TMR1L = 0xDC;

    /* initialise interrupts */
    INTCON = 0b11000000;
    PIE1 = 0b000000001;
    PIR1 = 0b000000000;
}

/* ===== */
/* INITIALISE LCD */
/* ===== */
void lcd_initialise(void)          /* Initialise LCD */
{
    /* tie the R/W pin low permanently, refer to the HD44780 d/sheet */

    unsigned char data, n;

    setbit(global, 1);          /* indicate 1x 4 bit transfer*/
    for(n=0; n<10; n++)
    {
        delay_10ms();          /* wait for power to stabilise*/
    }
    for (n=0; n<3; n++)
    {
        data = 0b00110000;
        lcd_write(data);
        delay_10ms();
    }

    /* function set */
    data = 0b00100000;
    lcd_write(data);
    delay_10ms();
}
```

## Simple C for the PIC Microcontroller

```
clearbit(global, 1);          /* indicate 2x 4 bit transfer */

data = 0b00101000;          /* N= 1; F=0 */
lcd_write(data);
delay_10ms();
/* display off */
data = 0b00001000;
lcd_write(data);
delay_10ms();
/* display clear */
data = 0b00000001;
lcd_write(data);
delay_10ms();
/* entry mode set */
data = 0b00000110;          /* I/D = 1 S = 1 */
lcd_write(data);
delay_10ms();
/* display on */
data = 0b00001100;
lcd_write(data);
delay_10ms();

}                             /* initialisation ends */
```

### lcd.c File

Here is the lcd.c file:

```
#include <header.h>
#include <htc.h>

/* ===== */
/* Split 'data' into upper and lower nibbles & present to port */
/* ===== */
void lcd_write(unsigned char data) /* 4 bit LCD transfer */
{
    load_lcd(data & 0xF0);          /* load top 4 MSBs */
    if ((global & 0b00000010)==0) /* 2 nibble transfer if bit clear */
    {
        data <<= 4;                /* shift LSBs to MSBs */
        load_lcd(data & 0xF0);      /* load bottom 4 MSB */
    }
    delay_10ms();
}

/* ===== */
/* Put data on port pins and set/clear Enable line */
/* ===== */
void load_lcd(unsigned char lcd_data) /* load data into LCD */
{
    setbit(PORTA, E);
```



## Arrays

```
PORTB &= 0x0F;           /* clear top 4 bits */
PORTB |= lcd_data;      /* toggle only DB7:DB4 */
asm("nop");
asm("nop");
asm("nop");
asm("nop");
clearbit(PORTA, E);     /* clock in data */
}
```

The function `lcd_write()` takes in an 8-bit variable and the statement

```
load_lcd(data & 0xF0);
```

clears the lower 4 bits of this variable in readiness for presenting this byte to the port. Since we are only using the upper 4 bits of Port B and do not want to change the lower 4 bits, we need to set the lower 4 bits equal to '0'. The function `load_lcd()` sets the Enable pin on the LCD high then clears the top 4 pins of Port B using the statement

```
PORTB &= 0x0F;
```

Note that we set the Enable line using the macro `setbit` defined in our header file. The data (with the lower 4 bits cleared) is then OR'd with Port B thus enabling us to only change the upper 4 bits of Port B, leaving the lower 4 bits unchanged.

The line

```
asm("nop");
```

is a 'No Operation' that effectively inserts a delay of a few instruction cycles between presenting the data to the port and taking the Enable line low using the line

```
clearbit(PORTA, E);
```

In our function `lcd_write()`, once we have sent the top 4 bits to the port, we then check the status of bit 1 in the `global` register. If bit 1 is cleared, we enter the `if` statement, shift the lower 4 bits into the place of the upper 4 bits and load them to the output port as before.

### display.c File

We are now going to examine the `display.c` file.

Here is `display.c`

```
#include <header.h>
#include <htc.h>

/* ===== */
/* Steps through array and puts characters onto the LCD */
/* ===== */
void show_row(unsigned char string[]) /* display row on lcd */
{
    unsigned char n;

    setbit(PORTA, RS); /* put into data mode */
    for(n=0; n<16; n++)
    {
        lcd_write(string[n]);
    }
    clearbit(PORTA, RS);
}
```

This function does nothing more than set the RS pin in Port A. This puts the LCD in data mode so it can accept characters instead of instructions. We then use a `for` statement to step through the array and the `lcd_write()` function to write the characters to the LCD display.

### delay.c File

Finally, we are going to discuss the `delay.c` file.

Here is the `delay.c` file

```
#include <htc.h>
#include <header.h>

/* ===== */
/* 10ms Delay */
/* ===== */
void delay_10ms(void) /* 10ms delay */
{
    TMR1L = 0x1E;
    TMR1H = 0xFB;
    setbit(T1CON, 0); /* start TMR1 */
}
```

## Arrays

```
while (!(global & 0b00000001))          /* has interrupt occurred? */
{
    asm("nop");
}
clearbit(global, 0);                    /* reset interrupt flag */
clearbit(T1CON, 0);                     /* delay has passed */
                                        /* stop TMR1 */
}
```

This is the same as before, but with different *hex* values loaded into the Timer 1 registers. We need a delay of 10ms. A clock cycle of 8us (using a prescaler of 1:8) implies that our count is calculated as follows

$$count = \frac{10ms}{8us} = 1250$$

We need to ensure Timer 1 overflows after 1250 cycles, so we have to preload Timer 1 with a value of

$$65536 - 1250 = 64286 = FB1E \text{ hex}$$

Once the files above have been added to our project and the Search Path has been set, the project can be compiled.

The program was downloaded to the PIC and operated perfectly. The power supply was intermittently connected to try to get the program to crash without success, implying that our LCD program is rock solid!

# Chapter 12

## Other Useful

# Snippets

In this chapter we will outline code techniques that you might need as your programming progresses.

### The `enum` keyword

The `enum` keyword provides a useful way of assigning integer values to a series of constants. You can then use the constant names instead of the numbers. The first constant in the brackets is automatically assigned a value of '0' unless specified otherwise. In the example below, we assign the value 1 to Monday. The constants that follow are then assigned a value of 1 greater than the constant before, so Tuesday is assigned the value 2, Wednesday the value 3 and so on. This can make your code a lot easier to read. An example is shown below.

```
enum days
    {Monday=1, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};

x = Monday + 1;
```

This returns the result for  $x$  as '2'.

### The `static` keyword

We have discussed functions already and we know that variables defined in a function are local to that function. We can define exactly the same variable in another function and the two variables can co-exist in the same program as completely separate entities. Once a function is exited, all variables defined in that function are cleared, thus when we return to that function, we have to reassign values to those variables and cannot pick up from where we left off. The variable has no memory once the function is left.

The `static` keyword overcomes this. Variables defined as `static` maintain their value between function calls.

In honesty, it is perfectly possible to program for years with the PIC and never have to use the `static` keyword, but we had to reference it for completion

### The `volatile` keyword

We have discussed that a compiler breaks down C code into assembly language and then into machine code. Since embedded applications (such as applications that run on the PIC microcontroller) need to run in small amounts of code space, C compilers really have a tough job in optimising the code to fit into a small code space.

Some compilers are cleverer than you think and if they see that a variable cannot be modified in code, they will construct assembly language to reflect this. An example of this is if a variable is defined, but never used, some C compilers will just ignore it removing it from the assembly code. The net effect of this is that some simulators (MPLAB included) will not show that variable in the Locals window. After all, why bother showing a variable that does not do anything?

Now, variables are not only modified by C code. A microcontroller interacting with the outside world can have its port pins and registers changed at any instant and you don't want a clever compiler thinking that, since your *code* is not modifying a variable, then that variable will *never* be modified. If there is a chance that a variable could be modified in any way aside from in code, then it should be declared as `volatile`. The `volatile` keyword tells the compiler that the variable might be altered by outside forces and it should not get too clever in optimising the code around that variable.

# Appendix A

# Variable Names

You have a large degree of freedom in what you call the variables in your program, although C does impose some restrictions. Firstly, it is advisable to name the variable according to the data that it holds in the program. Thus if you are writing a program to stores an address, calling the variable `address` instead of `var1` makes the code much more readable.

Variable names can be made up of letters and digits, but the first character of the variable name has to be a letter. Therefore `var1` is OK, but `1var` is not. You can use an underscore in the variable name too as this is classed as a letter, so if you need to store a memory address, the variable `memory_address` is perfectly valid. However, it is not advisable to use an underscore at the start of a variable name as the compiler sometimes generates its own internal variable names starting with an underscore. Variable names cannot contain spaces.

The latest versions of C allow you to use up to 31 characters in a variable name.

C makes the distinction between upper and lower case too, but the convention in many programming language states that variables are named with lower case and constants are named with upper case. Thus

```
unsigned char n;  
#define PI 3.1415
```

Naturally, C has its own set of keywords that you cannot use, many of these have been discussed so far. These are

<code>auto</code>	<code>double</code>	<code>int</code>	<code>struct</code>
<code>break</code>	<code>else</code>	<code>long</code>	<code>switch</code>
<code>case</code>	<code>enum</code>	<code>register</code>	<code>typedef</code>
<code>char</code>	<code>extern</code>	<code>return</code>	<code>union</code>
<code>const</code>	<code>float</code>	<code>short</code>	<code>unsigned</code>
<code>continue</code>	<code>for</code>	<code>signed</code>	<code>void</code>
<code>default</code>	<code>goto</code>	<code>sizeof</code>	<code>volatile</code>
<code>do</code>	<code>if</code>	<code>static</code>	<code>while</code>

# Appendix B

## C Operators and their Shortforms

The table below shows all the C operators with a description.

<b>Operator</b>	<b>What it does</b>
=	Assigns a value to a variable**
+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulus *
>	Is greater than
>=	Is greater than or equal to
<	Is less than
<=	Is less than or equal to
==	Compares a variable with a value**
!=	Is not equal
&&	Logical AND
	Logical OR
!	Logical NOT
++	Increment
--	Decrement
&	Bitwise AND
	Bitwise OR
^	Bitwise exclusive OR
<<	Move bits to the left
>>	Move bits to the right
~	Invert all bits

\* The modulus of a number is the remainder when one number is divided by another.

\*\*A distinction has to be made between an equals sign (=) and a double equals sign (==).

The single equals sign is an *assignment*. For example:

If A = 10 and B= 20 then

A = B

assigns the value of 20 to A (and keeps B unchanged).

The operator == is a comparison, so the phrase

A == B

compares A with B (and in this case declares the comparison to be false since A is not equal to B).

Here is a simple example to help explain:

```
A = 0;
B = 10;
while(1)
{
    A++;
    if(A == B)
    {
        A = 0;
    }
}
```

A is assigned the value 0 and B the value 10 using the single '=' sign (the assignment operator). The code inside the brackets belonging to the `while(1)` statement loops forever. A is compared to B using the comparison operator '==' in the line

```
if(A == B)
```

If A is equal to B, then A is reset to zero. If A is not equal to B the statement is ignored.

Shortforms of some of the above operators are often used. The table below shows examples of the shortforms with a description.



## Appendix B: C Operators and their Shortforms

<b>Operator</b>	<b>Shortform</b>	<b>Means</b>
+	a += 3	a = a + 3
-	b -= a	b = b - a
*	c *= 2	c = c * 2
/	d /= a	d = d/a
%	e %= 2	e = e % 2
++	x++	x = x + 1
--	x--	x = x - 1
<<	y <<= 3	y = y << 3
>>	z >>= 2	z = z >> 2

# Appendix C

## The #define

### Directive

We said that C is a language that is designed to be readable and many programmers go to extraordinary lengths to make their code as near to conventional English as possible. The `#define` statement allows this to happen.

The `#define` statement allows the programmer to replace certain lines of code with something more readable.

In the code below, we are replacing the statement on the far right with the statement in the middle.

```
#define setbit7 PORTB = (PORTB | 0b10000000)
```

Thus every time the compiler sees the code

```
setbit7
```

it replaces it with the code

```
PORTB = (PORTB | 0b10000000)
```

Note: the define statement does not have a semicolon at the end.

Nearly all C programs use this trick to make the code much easier to read. The most common usage of the `#define` statement is when setting, clearing and checking bits as follows

## Appendix C: The `#define` Directive

```
#define setbit(address, bit) (address = address | (1<<bit))
#define clearbit(address, bit) (address = address & ~(1<<bit))
#define checkbit(address, bit) (address & (1<<bit))
```

It is useful to always have these lines in a generic header file that you use in all programs.

Therefore the line

```
setbit(PORTB, 1);
```

can be used to set Port B bit 1 instead of

```
(PORTB = PORTB | (1<<1));
```

# Appendix D

# Suggested LED

# Schematics

Below are shown suitable schematics for the LED experiments in this book.

The PIC16F627A is powered from 5V. An LED needs a voltage across it to turn it on (about 1.8V) and when it is lit it needs about 15mA to light brightly. Assuming the port pins output 5V when high and each LED has 1.8V across it, this means that each resistor (R1 – R8) has 3.2V across it. From Ohm's Law, where R is resistance, V is voltage and I is current

$$R = \frac{V}{I}$$

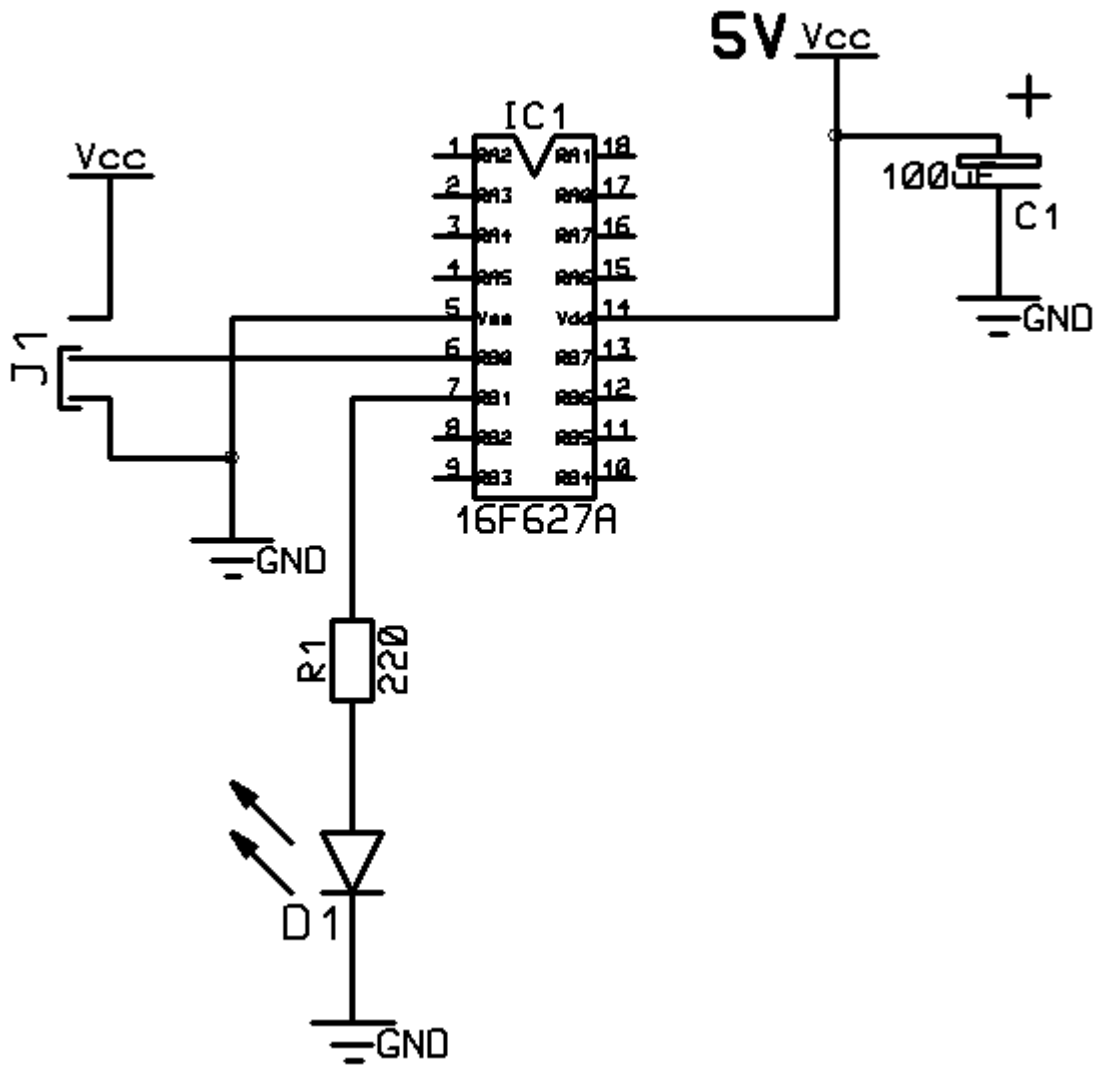
implying that a resistance of

$$R = \frac{3.2}{0.015}$$

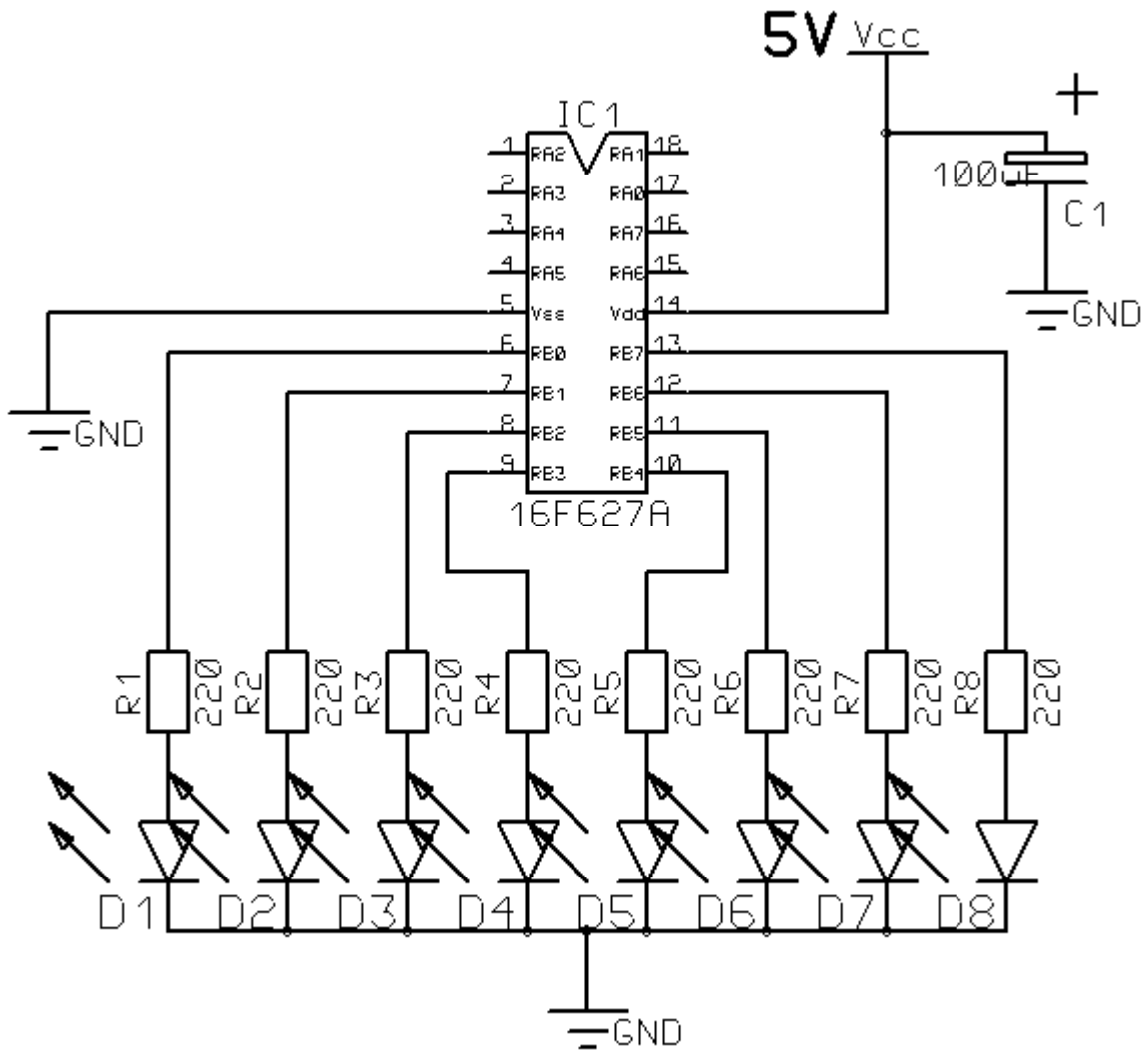
is needed, meaning our resistor needs to be 213 Ohms. Therefore, we use a 220 Ohm resistor.

Capacitor C1 is 100uF, 6.8V electrolytic and is only there to smooth any noise that might appear on the power supply rail from the LEDs switching.

Appendix D: Suggested LED Schematics



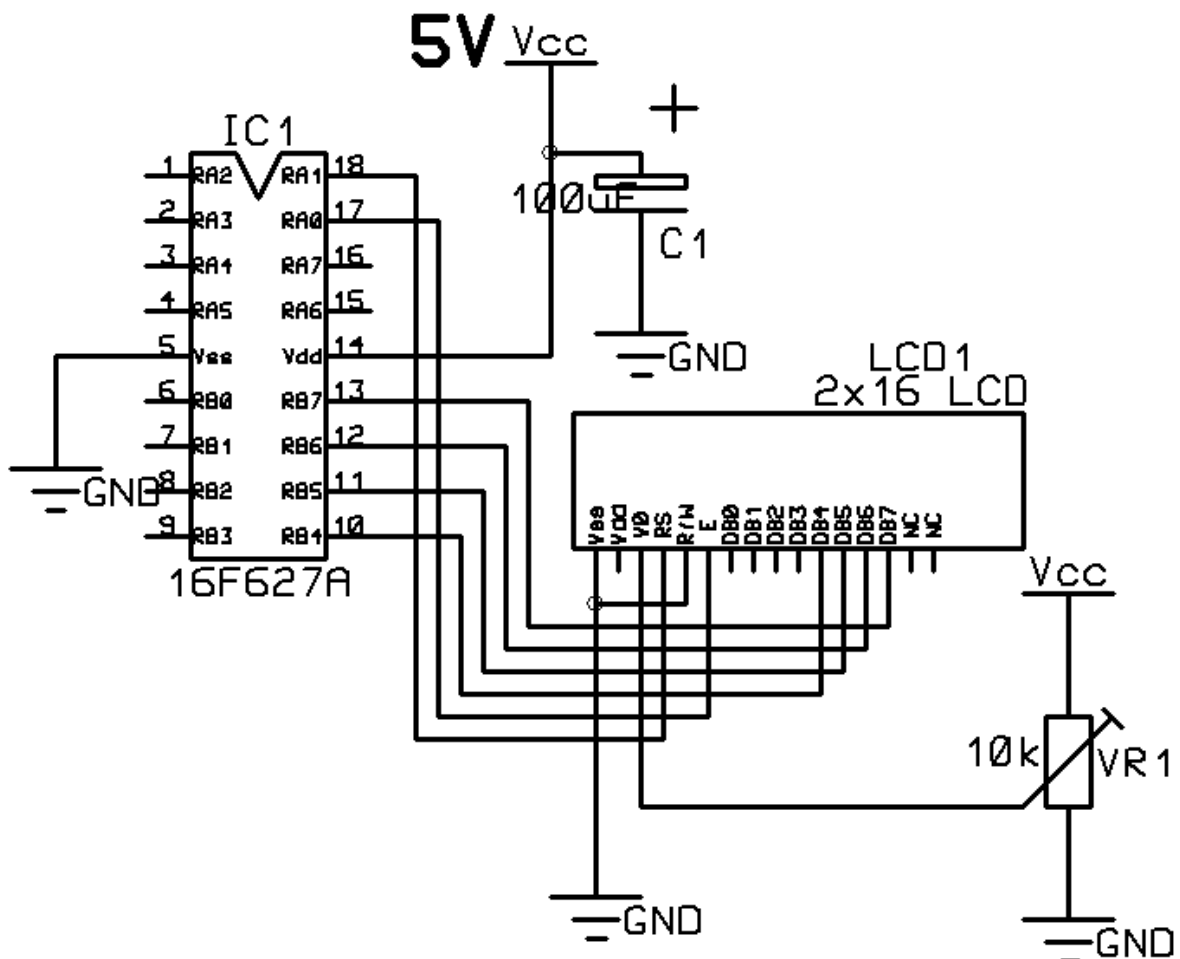
Single LED Circuit



**8 Way LED Circuit**

# Appendix E Suggested LCD Schematic

The schematic for the LCD display is shown below. The Vcc line (5V) has been omitted for clarity. Pins RB7 – RB4 connect to DB7 – DB4 respectively. RA0 connects to the Enable pin of the LCD and RA1 connects to the RS pin.



A 10k variable resistor can be used to adjust the contrast (by adjusting the voltage on the V0 pin from 0 to 5V).

# Appendix F

# Preprocessor Directives

Directive	Meaning	Example
#	Preprocessor null directive, do nothing	#
#assert	Generate error if condition false	#assert SIZE > 10
#asm	Signifies the beginning of in-line assembly	#asm MOVLW FFh #endasm
#define	Define preprocessor macro	#define SIZE 5 #define FLAG #define add(a,b) ((a)+(b))
#elif	Short for #else #if	see #ifdef
#else	Conditionally include source lines	see #if
#endasm	Terminate in-line assembly	see #asm
#endif	Terminate conditional source inclusion	see #if
#error	Generate an error message	#error Size too big
#if	Include source lines if constant expression true	#if SIZE < 10 c = process(10) #else skip(); #endif
#ifdef	Include source lines if preprocessor symbol defined	#ifdef FLAG do_loop(); #elif SIZE == 5 skip_loop(); #endif
#ifndef	Include source lines if preprocessor symbol not defined	#ifndef FLAG jump(); #endif
#include	Include text file into source	#include <stdio.h> #include "project.h"
#line	Specify line number and filename for listing	#line 3 final
#nn	(Where <i>nn</i> is a number) short for #line <i>nn</i>	#20
#pragma	Compiler specific options	Refer to <b>Section 3.10.3 "Pragma Directives"</b>
#undef	Undefines preprocessor symbol	#undef FLAG
#warning	Generate a warning message	#warning Length not set



# Appendix G

# ASCII Codes

Code	Character	Code	Character	Code	Character	Code	Character
32	(space)	62	>	92	\	122	z
33	!	63	?	93	]	123	{
34	"	64	@	94	^	124	
35	#	65	A	95	_	125	}
36	\$	66	B	96	`		
37	%	67	C	97	a		
38	&	68	D	98	b		
39	'	69	E	99	c		
40	(	70	F	100	d		
41	)	71	G	101	e		
42	*	72	H	102	f		
43	+	73	I	103	g		
44	,	74	J	104	h		
45	-	75	K	105	i		
46	.	76	L	106	j		
47	/	77	M	107	k		
48	0	78	N	108	l		
49	1	79	O	109	m		
50	2	80	P	110	n		
51	3	81	Q	111	o		
52	4	82	R	112	p		
53	5	83	S	113	q		
54	6	84	T	114	r		
55	7	85	U	115	s		
56	8	86	V	116	t		
57	9	87	W	117	u		
58	:	88	X	118	v		
59	;	89	Y	119	w		
60	<	90	Z	120	x		
61	=	91	[	121	y		