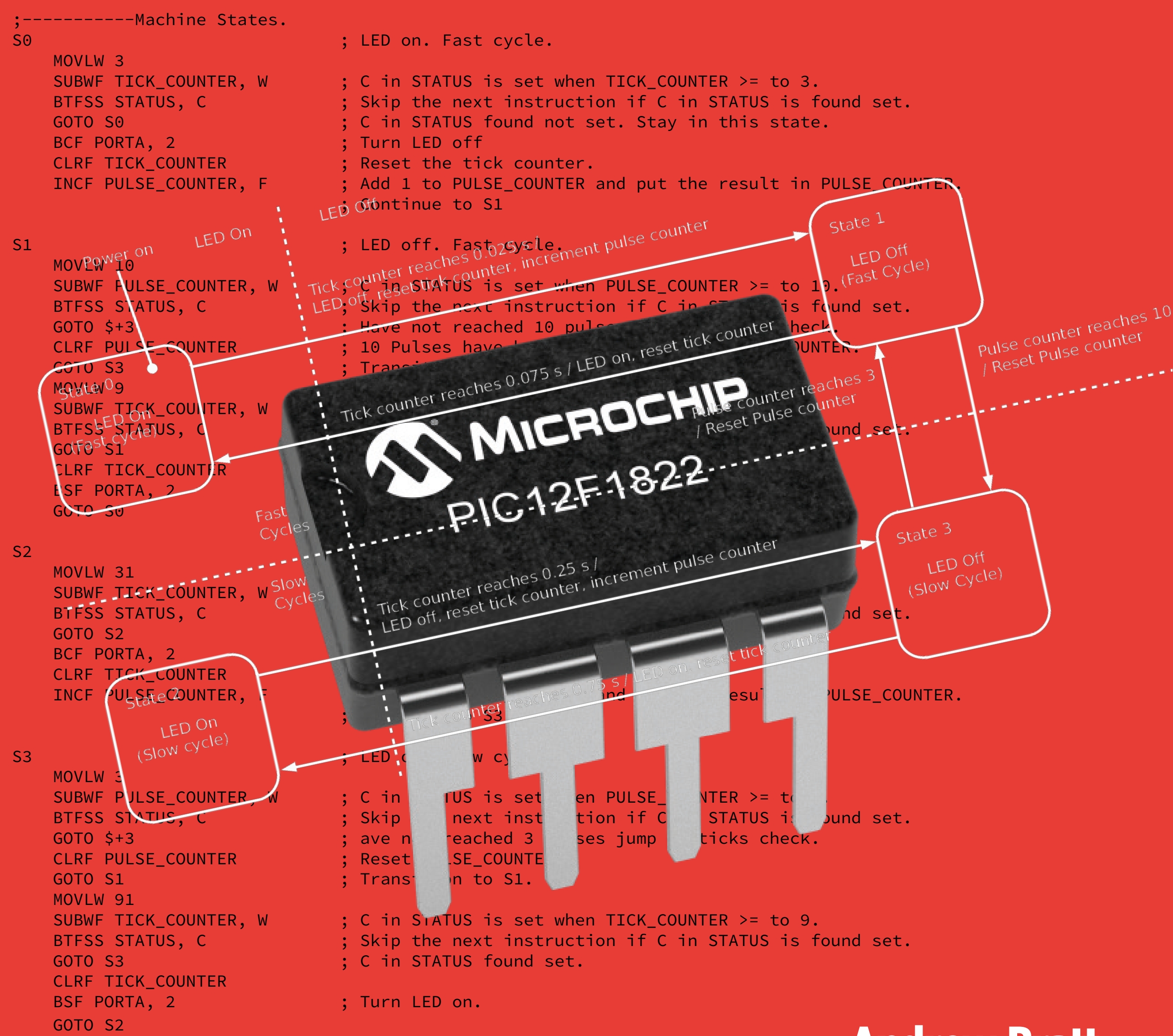
Programming the Finite State Machine with 8-Bit PICs in Assembly and C



Andrew Pratt

elektor
LEARN DESIGN SHARE

Programming the Finite State Machine with 8-Bit PICs in Assembly and C

Andrew Pratt

an Elektor Publication

LEARN DESIGN SHARE

● This is an Elektor Publication. Elektor is the media brand of
Elektor International Media B.V.
78 York Street
London W1H 1DP, UK
Phone: (+44) (0)20 7692 8344

First published in the United Kingdom 2020

● British Library Cataloguing in Publication Data
Catalogue record for this book is available from the British Library

● **ISBN: 978-1-907920-92-9**
EISBN: 978-3-89576-356-4
EPUB: 978-3-89576-357-1

Prepress production: DMC ¦ dave@daverid.com

Printed in the Netherlands by Wilco

Table of Contents

• Preface

This practical guide is aimed at electronics students and hobbyists. It is intended to be a valuable aid in writing programs using Finite State Machines (FSMs) in assembly language using 8-bit PIC microcontrollers. The last two chapters introduce the use of the C programming language and make a direct comparison with development in Assembly.

An FSM is a way of writing a program to make it easier to produce and modify. The machine is abstract in that it is just the structure of the program. This abstract machine can be represented by drawing a diagram on paper. The diagram is independent of the programming language used. The FSM chart gives a complete description of what the program does. It can then be implemented as source code.

The book should appeal to those with an interest in the combination of electronics and software and have an interest in how things work. The book will describe writing code for two particular microcontrollers: The 12F1822 and 16F1823. Both are mid-range and inexpensive. To read and write the programs to and from the PICs, all that is required is an FTDI TTL level USB lead (TTL-232R-5V-WE) in addition to two programs that are both available for free download as executable files and source code from Elektor. Microsoft Windows or Linux can be used.

The PIC programs are written in assembly language. This goes against the conventional wisdom of using a higher-level language such as C. One reason for this is that assembly is a good way of learning what is happening at the lowest level. This is important as microcontroller programming requires an understanding of the chip. Another reason for using the finite state machine approach is that it makes assembly programs surprisingly easy to follow. One of the main obstacles in the way of getting started with embedded programming is the installation and learning of new software tools.

The emphasis of this book is on making things straightforward with as little complication as possible. Therefore you can concentrate on understanding the code. Real projects aren’t just about coding: our software has to do something real. As a consequence, a chapter deals with a method of circuit board construction.

All coding is done in a text editor of your choosing. The command line is used for running programs. If you are a Windows user, you might look at this as old fashioned. This is actually an efficient way of doing things: simple scripts for repetitive tasks save lots of mouse clicks. The last two chapters give an introduction to programming in C using the XC8 compiler. Again this is done using a text editor and the command line.

The intention has been to achieve results using an inexpensive microcontroller with simple command line tools. Much emphasis is placed on using Microchip’s datasheet as this is the best place to get correct detailed information.

Chapter 1 • Getting Started

1.1 • Introduction

Don't be put off by the technical-sounding words 'Finite State Machine' (FSM). The principle of how they work is really easy to understand. Firstly, the machine is not an actual machine that you have to build. It is merely a structure your program has. The idea is that the program is broken down into a set number of states. When the program is in one of these states, it can respond to a set of inputs or events resulting in the program jumping to a different state. The outputs from the program depend only on the state (Moore machine) or on the state and the inputs (Mealy machine). Let's start with the basic 'Hello World' microcontroller program to flash an LED as an example of a state machine with two states, LED ON and OFF.

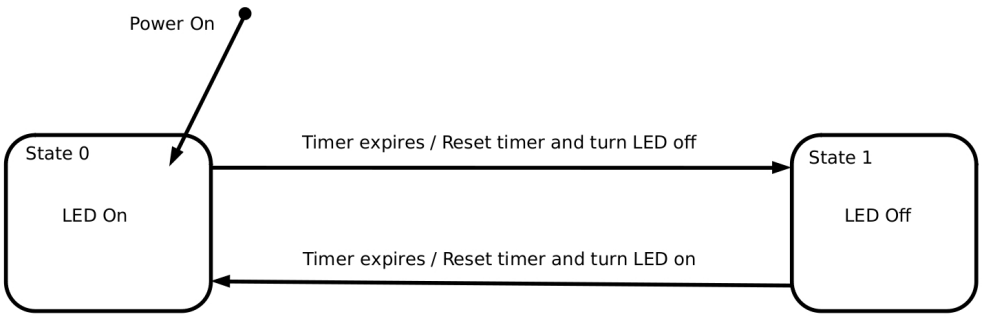


Figure 1-1 A two-state machine

Figure 1-1 displays what the program does and how to write it.

What the program does is:

After power up, the initial conditions are the LED is ON and the timer is reset and started.
After the timer expires, the LED turns OFF and the timer is reset and started.
After the timer expires again, the LED turns ON and the timer is reset and started.
This continues until the microcontroller is powered down

In this example, there is only one event that triggers a change of state: This being the expiry of the timer. The effect of the expiry depends on the current state - it might jump to the LED ON state or the LED OFF state. The fact that an input or event is only used by the current state helps to make a more complicated program robust and easier to follow. This example is a Mealy machine, where outputs or actions are determined by the current state and inputs. On the state diagrams, the transitions between states are shown by arrows. Alongside these are the input conditions that trigger transitions and the resulting outputs. In a Moore machine, the outputs depend only on the current state of the machine. The outputs would be shown on the diagram inside the states. This book is specifically about practical programs. I will not try to adhere to strict definitions. As for how

to write the program, you have already done the important bit by planning. All that has to be done now is to implement the diagram as code and this will be less prone to error as you can concentrate on local detail. The book will mostly use the Mealy type machine. Inputs causing a transition will be separated by a forward slash from the outputs. This will be written next to the transition between two states. With regards to the FSM, inputs and outputs (I/O) are not restricted to electrical I/O. They also include other types of change. An input could be the timer timing out or an output could be the writing of a value to a register in the CPU. FSMs can be written in just about any programming language.

1.2 • Practical Implementation

Microcontroller projects do not need to be expensive. The two PICs that are going to be used throughout the book are the 12F1822 and 16F1823 variants. Both are cheap mid-range devices, costing less than one pound in the UK at the time of writing. They are available in the dual in-line package (DIP) form that is easy to solder or can be plugged into a holder. The programs will be written in a text editor and assembled with the gpasm open-source assembler. To load your programs into the PIC, you could use a programmer such as PICKIT 3, but there is no need to buy one. Two programs have been written for this book that can read and write using an FTDI USB lead *TTL-232R-5V-WE*. This same lead can also be used for reading and writing data from PICs in later projects.

1.2.1 • Choice of Operating System

Microsoft Windows or Linux can be used for assembling and loading programs to the PICs. There are two archive files available by way of a free download from the Elektor website: One is for Windows and the other for Linux. These files contain all the source files for the examples in the book and applications mentioned above. As described in the preface, the command line method is used throughout the book for both Linux and Windows. The use of the terminal with typed commands might seem like a throwback to the last century for those who remember using DOS. It is, however, a very easy way of doing things, especially for repetitive tasks like running assembler programs or loading the PIC with its program. Tasks can be simplified with scripts to save typing and recent commands can be recalled using the up and down keys. If you are using Linux, typing history will give a numbered list of the previous commands you have used. They can be run again by using the exclamation mark followed by the number. Auto-completion allows you to use the tab key to list possible files. You never have to type the complete name. If you are happy writing source code for an electronic device using text files then using the command line should be the logical choice. Graphic User Interfaces (GUIs) have their advantages. However, if you are doing something repetitive, a lot of mouse clicking or even using keyboard shortcuts is tedious compared with running a script or recalling a previous command and its options.

1.2.2 • Machine Language and Assembly Source Files

The PIC microcontroller has to be programmed by loading the machine code into its flash memory. The word ‘programmed’ in the last sentence means transferring machine code from a file on a computer to the PIC as opposed to writing the code. The machine code is

the raw byte values that it processes when running. This machine code is first assembled into a file called a hex on your computer. It gets its name from the Intel Hex Format which is a file standard for this purpose. The hex file is then written to the PIC using a combination of a program on your computer and a hardware interface called a programmer. When referring to machine code, this is not the finite state machine but the processor itself being referred to as a machine.

It is possible to write your program by typing a hex file directly into a text editor. Manually assembling machine instructions is too time-consuming to be considered a realistic task. To make it easier, code is written in assembly language where short acronyms for machine instructions are used to produce more readable code. The file that you type this way is the source code and usually has the file suffix .asm. This file is then read by the assembler application and written to the hex file as raw machine code.

It is generally accepted that it is better to write your programs in a higher-level language. If you want to get things done quicker and with less chance of error, then this is absolutely true. C language was invented to get away from the problems of programming in assembly language. This book is about finite state machines implemented on simpler PICs. To write code for a microcontroller, you have to understand what is going on at the level where software meets hardware. Where C makes coding faster and easier, it also makes things more complicated. You don't, however, see this extra complication: it's taken care of by the compiler and linker. All of this gets in the way of understanding what your code is doing. For the PICs used in this book, there are 49 available instructions. The ones commonly used are quickly learnable. The beauty of assembly language is that you can see what's going on. The code you write is exactly the machine code that goes into the PIC. Once the source code file is finished, it is converted to machine language by the assembler application: in our case the program *gpasm*. The assembler reads your source code and outputs the hex file. The hex file is then read by another application that writes to the PIC using some form of hardware interface. Figure 1-2 below depicts this programming chain.

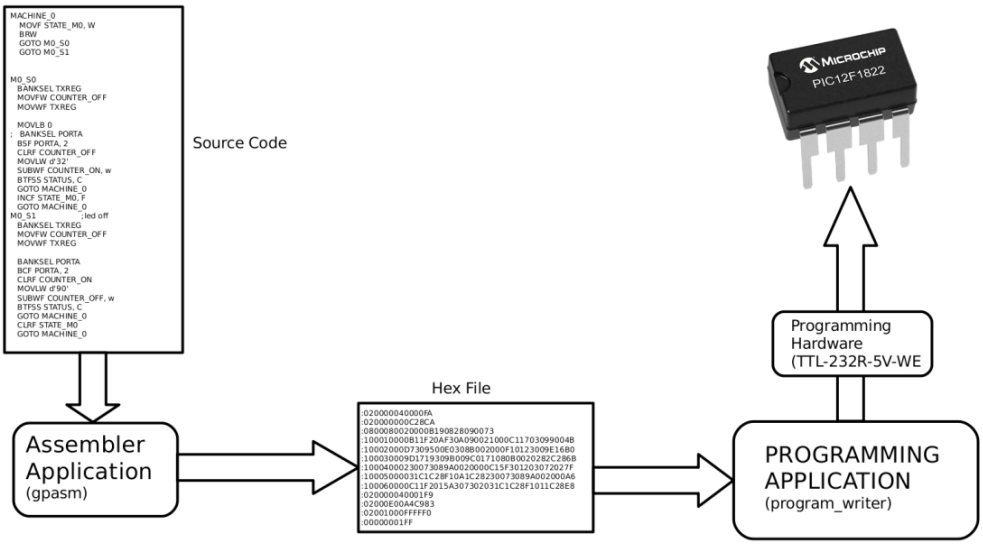


Figure 1-2 Programming Chain

You can if you prefer, use MPLABX and PICKET 3 or 4 from Microchip. You may already have these and be familiar with them. The next two paragraphs describe an alternative using the applications provided in the download.

1.2.3 ● Using the Applications on Microsoft Windows

First, you need to download and install gputils for Windows from here:
https://sourceforge.net/projects/gputils/files/gputils-win32/

The version used at the time of writing was 1.4.0. Although not the latest version, it was chosen to be the same as used by the current Debian and Ubuntu. You can use the latest version if like.

You will need the FTDI driver for your USB interface lead, and the installation of the file FTD2XX.DLL. The best way to do this is to download the executable setup for Windows from:
http://www.ftdichip.com/Drivers/D2XX.htm

At the time of writing, it is named CDM2128_Setup.zip. Inside the zip file is *CDM2128_Setup.exe* that will install the driver and the required DLL file. Install it before plugging in the lead.

You need to unzip the file from the Elektor website into a convenient directory on your computer. Please make sure that you download the correct file. The Linux version won't work on Windows and vice versa. It's not just executables that are not compatible. Text files have different line endings: Linux files have a new line character while Windows files

have a carriage return and new line character at the end of each line. The file unzips several directories. The working directory contains a batch file called terminal_here. Clicking on this will open a virtual terminal. If you type dir and press return, the contents of the directory will be displayed. Initially, the files in the working directory are executable. These executables will be run from the command line. This only involves typing a couple of words and pressing return. Later in this chapter, instructions will be given on how to use these applications. For reading and writing PIC source files, you can use Windows' text editor, notepad, but any text editor will do, including *notepad*++.

1.2.4 • Installing the Applications on Linux

The archive file for Linux is a tar file. This can be extracted by typing on the command line *tar -xvf <file name>.tar.*

There are many variants of Linux. I won't try to cover the instructions in detail for all of them. I will give instructions for two: Debian and Fedora. The details for Debian are also applicable to Ubuntu and other Debian derivatives.

The assembler program *gpasm* is part of the GNU PIC Utilities (*gputils*) package. This can be installed on Debian based Linux by opening a terminal window and typing:
sudo apt-get install gputils.

Of course, you have to be connected to the Internet. On Fedora, the command is *sudo dnf install gputils*.

```
user@debian:~/programs/chap01_progs$ sudo apt-get install gputils
```

Figure 1-3a Installing gputils on Debian (Ubuntu etc)

For Fedora:

```
[user@localhost ~]$ sudo dnf install gputils
```

Figure 1-3b Installing gputils on Fedora

To load the hex file contents into the PIC microcontroller and read back the loaded bytes, the two programs that have been written especially for this book are called *program_writer_xx* and *program_reader_xx*, where *xx* is 32 or 64. Both versions are provided: one for 32-bit and one for 64-bit operating systems. These programs need to have the package libftdi1-dev installed to work. This can be installed on Debian by opening a terminal window and typing *sudo apt-get install libftdi1-dev* .

```
user@debian:~/programs/chap01_progs$ sudo apt-get install libftdi1-dev
```

Figure 1-4a Installing libftdi1-dev on Debian

For Fedora:

```
[user@localhost ~]$ sudo dnf install libftdi
```

Figure 1-4b Installing libftdi on Fedora

These two applications have only been tested with these two PICs. Figure 1-5 shows the directory trees for the Linux and Windows downloads.

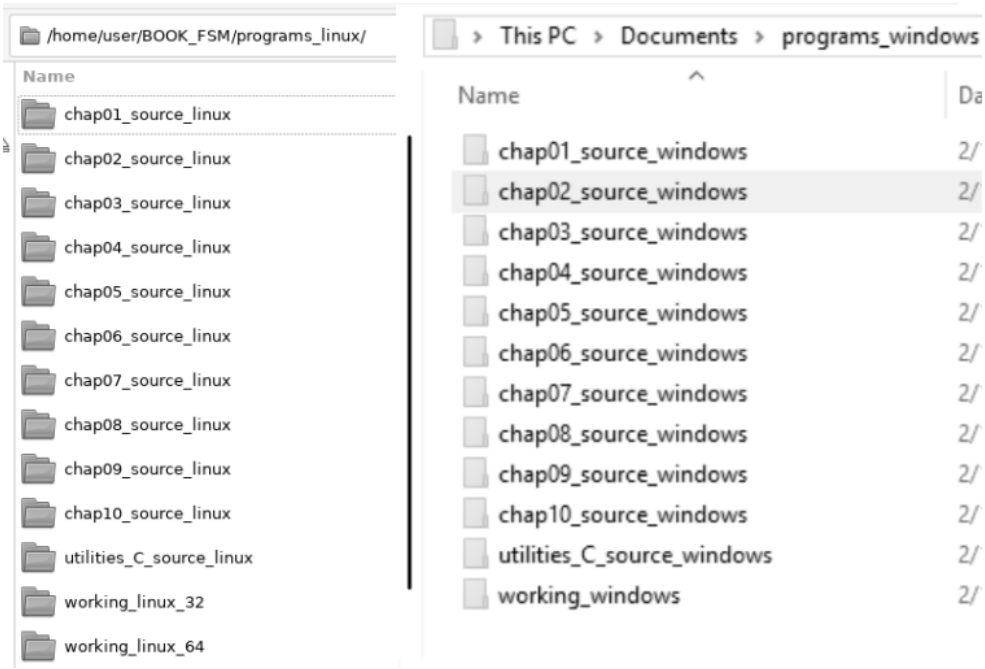


Figure 1-5 Download directory tree for Linux and Windows

Once you have all the software installed, the next thing is to try it out using the hardware.

1.2.5 ● The FTDI Lead and Testing the Programming Chain

All that is required is an FTDI *TTL-232R-5V-WE* USB wire end lead. They are readily available from suppliers such as Farnell, RS Components, and Digi-Key. 1k resistors to pins 4, 6, and 7 are recommended for protection. The one connected to pin 5 is to limit the current flowing through the LED.

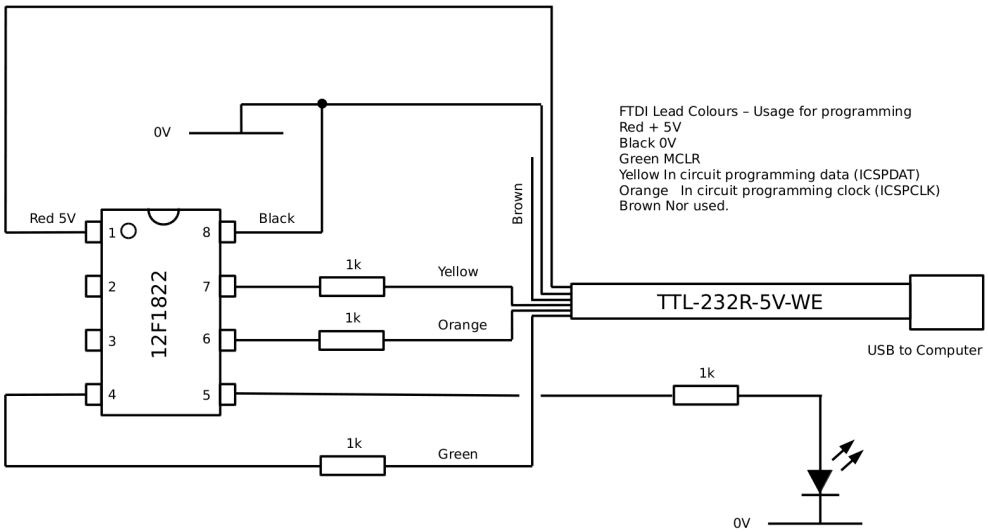


Figure 1-6 In-Circuit Programming Diagram

The above circuit can quickly be connected to a plugin breadboard: see Figure 1-9. The LED is not part of the programming circuit. It is there for the test program to demonstrate that it is all correctly working. The test program *test.hex* is included in the *Chap01_progs* folder of the archive file. It makes the LED flash on for a quarter of a second and off for three-quarters of a second. You need to copy it to the working directory to use it.

If you are using Windows, click on the *terminal_here* batch file in the working directory to open the terminal window. Type in the following and press return. See the upper pane in Figure 1-7.

```
program_writer.exe test.hex
```

If you are using Linux, open a virtual terminal and navigate to the working directory. Type the following and press return. See the lower pane in Figure 1-7.

```
sudo ./program_writer_64  test.hex
```

Note that in Linux you must prefix the executable file with ./ .This is the path to the present directory. The *sudo* command will be required to acquire the privilege of opening the USB port. Once the code is loaded into the PIC, the LED should start flashing. The code can be verified by reading it back using the program *program_reader*: refer Figure 1-8. Only a few lines of the output from these programs is shown in the screen dumps. Take care to correctly connect the wires and get the LED the correct way round. The PIC has pin 1 marked with a dimple on the top surface.

```
C:\Users\Andrew\Documents\FSM\working_windows>program_writer.exe test.hex
address = 0, data from file = 280b
address = 1, data from file = 0
address = 2, data from file = 0
address = 3, data from file = 0
address = 4, data from file = 20
address = 5, data from file = 190b
address = 6, data from file = 2808
address = 7, data from file = 9
address = 8, data from file = 110b
address = 9, data from file = af0
address = a, data from file = 9
```

WINDOWS

```
user@debian:~/programs_linux/working_64$ sudo ./program_writer_64 test.hex
[sudo] password for user:
address = 0, data from file = 280b
address = 1, data from file = 0
address = 2, data from file = 0
address = 3, data from file = 0
address = 4, data from file = 20
address = 5, data from file = 190b
address = 6, data from file = 2808
address = 7, data from file = 9
address = 8, data from file = 110b
address = 9, data from file = af0
address = a, data from file = 9
```

LINUX

Figure 1-7 Writing the Hex File to the PIC: Windows and Linux 64 bit

```
C:\Users\Andrew\Documents\FSM\working_windows>program_reader.exe test.hex
address = 0  value from file / microcontroller = 280b / 280b
address = 1  value from file / microcontroller = 0 / 0
address = 2  value from file / microcontroller = 0 / 0
address = 3  value from file / microcontroller = 0 / 0
address = 4  value from file / microcontroller = 20 / 20
address = 5  value from file / microcontroller = 190b / 190b
address = 6  value from file / microcontroller = 2808 / 2808
address = 7  value from file / microcontroller = 9 / 9
```

WINDOWS

```
user@debian:~/programs_linux/working_64$ sudo ./program_reader_64 test.hex
[sudo] password for user:
address = 0  value from file / microcontroller = 280b / 280b
address = 1  value from file / microcontroller = 0 / 0
address = 2  value from file / microcontroller = 0 / 0
address = 3  value from file / microcontroller = 0 / 0
address = 4  value from file / microcontroller = 20 / 20
address = 5  value from file / microcontroller = 190b / 190b
address = 6  value from file / microcontroller = 2808 / 2808
address = 7  value from file / microcontroller = 9 / 9
```

LINUX

Figure 1-8 Reading Back PIC Code: Windows and Linux

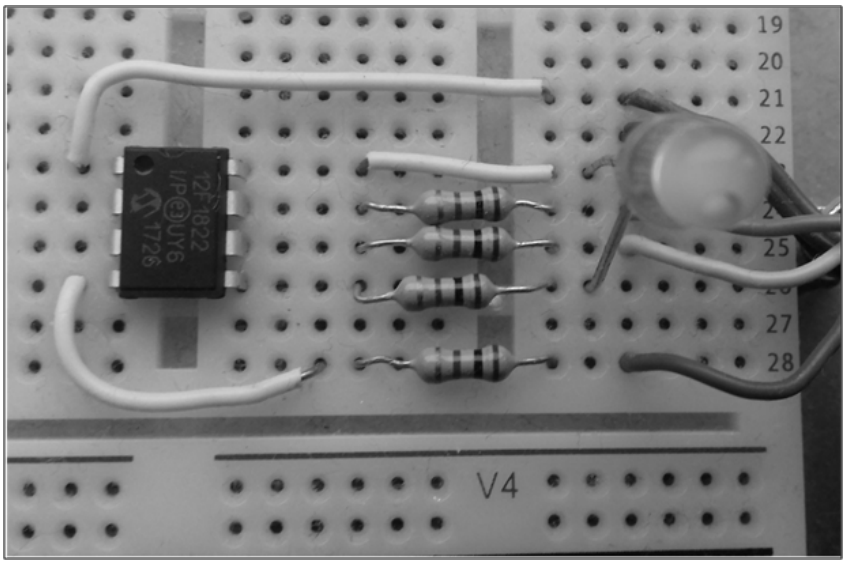


Figure 1-9 Plugin breadboard

1.3 • Some Fundamentals

Assuming that everything has so far worked, you can move on in the next chapter to writing a first PIC program with confidence that it can be written to the controller. The rest of this chapter is of benefit to those who are not familiar with PIC programming in assembly language.

All that is required to write the code is a text editor. You should also have open a copy of the datasheet for the 12F1822 and 16F1823 which can be downloaded free as a PDF file from Microchip. The name of the document is *DS40001413E*. To get you on your way, I will present a program that flashes an LED at about 5Hz. This is not the same program as test.hex that was used to prove the programming chain above.

Writing code for a microcontroller requires knowledge of the inner workings of a device. The controller is part of an electronic circuit. The code is an extension of the circuit that you are building. If you browse through the datasheet, you'll see that there are more than four hundred pages. Trying to read a document like this can be overwhelming as it has not been written as a teaching aid and is packed with information. We will start slowly with just enough detail to start-off. We will add more as the projects get more advanced. Once you start to become familiar with the layout of this manual, it is quite easy to find what you want to know as you progress. I'm assuming that all of this is new to you and will try to explain all you need to know to understand this first program.

1.3.1 • Bits and Bytes

Let's start with the bit. The word bit is the contraction of binary digit. A bit is a basic unit of

information: it has only two states commonly referred to as 0 or 1. Physically these states can be represented by say a voltage of 0V or +5V. Any physical quantity that can have two distinct values could be used. Now if two bits are placed next to each other, there are now four permutations that bits can be in, 00, 01, 10, 11.

If three bits are used, there are eight permutations, 000, 001, 010, 011, 100, 101, 110, 111. Whenever a bit is added to the group, the number of permutations double. A byte is normally a group of eight bits and these have 256 permutations. Each arrangement of bits can be used to represent an integer. To start with we'll look at positive integers and zero. Negative integers can also be expressed but that's for later.

Bits that are 1 are sometimes referred to as being on, and off when they are 0.
Each bit in a byte not only has a value of 0 or 1 but has a position in the byte. We give weight to the positions so that the bit on the right end is worth 1 when it is on and 0 when it is off. The next position has a weight of 2 so that bit is worth 2 when it is on and 0 when it is off. Continuing this binary weighting to all eight bits in a byte, all the values from 0 to 255 can be allocated to bit states.

	128	*64*	*32*	*16*	*8*	*4*	*2*	*1*
0	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
1	*0*	*0*	*0*	*0*	*0*	*0*	*0*	**1**
2	*0*	*0*	*0*	*0*	*0*	*0*	**1**	*0*
3	*0*	*0*	*0*	*0*	*0*	*0*	**1**	**1**
4	*0*	*0*	*0*	*0*	*0*	**1**	*0*	*0*
5	*0*	*0*	*0*	*0*	*0*	**1**	*0*	**1**
6	*0*	*0*	*0*	*0*	*0*	**1**	**1**	*0*
7	*0*	*0*	*0*	*0*	*0*	**1**	**1**	**1**
8	*0*	*0*	*0*	*0*	**1**	*0*	*0*	*0*
9	*0*	*0*	*0*	*0*	**1**	*0*	*0*	**1**
10	*0*	*0*	*0*	*0*	**1**	*0*	**1**	*0*
11	*0*	*0*	*0*	*0*	**1**	*0*	**1**	**1**
12	*0*	*0*	*0*	*0*	**1**	**1**	*0*	*0*
13	*0*	*0*	*0*	*0*	**1**	**1**	*0*	**1**
14	*0*	*0*	*0*	*0*	**1**	**1**	**1**	*0*
15	*0*	*0*	*0*	*0*	**1**	**1**	**1**	**1**
16	*0*	*0*	*0*	*1*	*0*	*0*	*0*	*0*
-	-	-	-	-	-	-	-	-
255	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

Table 1-1 Binary Weighting