

My First PIC Projects

An introduction to the PIC processor.

mICro's First Projects

[Introduction](#)

[Flash That LED](#)

[Define The Problem](#)

[Writing The Software](#)

[Mnemonics](#)

[The Assembler](#)

[Labels](#)

[Using a text assembler](#)

[The Flowchart](#)

[MicroPlan Assembler](#)

[The Simulator](#)

[The Real World](#)

[7 segment LED display](#)

[The Counter](#)

[Using Switches](#)

Introduction

Welcome to **MICRO's**. I know you are itching to get started with your new software and begin programming PICs as soon as possible, so this introduction will get you familiar with the **MICRO's** suite of software and during the process you will also start writing some small programs and hopefully get them working in the real world.

If you have the **MICRO's** Experimenters Kit, then going through the projects will be quite easy. If you do not have the Experimenters Kit, then you may like to get a PIC programmer so that you can program the software into a 16F84 chip to complete the projects.

Before we get going, you have to understand that a PIC, or any other microcontroller chip for that matter, is just a piece of silicon wrapped in plastic with pins sticking out to connect it to the outside world. It does not have any brains, nor can it think for itself, so anything the chip does is the direct result of our intelligence and imagination. Sometimes you may get the feeling that these things are alive and are put here to torment your every waking minute, but this is usually due to bugs in your software, not a personality inside the chip. So please remember:

The PIC will always do what you tell it to, not necessarily what you want it to.

One other thing that can cause problems is in the way you handle the chip itself. Your body is more than likely charged with Static Electricity and is usually the zap you feel when you touch a metal object after walking on nylon carpet or similar. The PIC's most definitely do not like this high voltage discharging into them. It can destroy the functionality of the chip either totally or partially, so always try to avoid touching the pins with your fingers.

The PIC 16F84 data sheet is available in PDF format on the CD ROM in the Acrobat directory.

Flash That LED

This would have to be the universal number one project for new PIC programmers. If you have had anything to do with writing software for PC's, then it would be the equivalent of writing "hello world" on the monitor for the first time.

You might be thinking at this stage..."What a boring project. I want to create a robot that does amazing things, not mess around with silly 'hello world' or LED flash programs."

Patience my friend. Things like that will come in due course, and as the old saying goes, "You have to crawl before you can walk".

OK then, so how do we get started?

You might be tempted to jump straight in and write volumes of code right from the start, but I can only say, that in all probability, your software will not work. Now this might sound a bit tedious, but "planning" is the best way to begin any piece of new software. Believe me, in the long run, your code will stand a much better chance of working and it will save you valuable time. Other benefits are that your code will be structured and documented much better, which means you can read through and understand it more easily in the future if the need arises.

So just how do we get this piece of silicon to do our bidding? In this case - flash a LED.

Fundamentally, the PIC needs three things to make it work.

- 1) 5 volt power source.
- 2) Clock source
- 3) Software

The 5 volt supply is there to power the chip. The clock source gives the chip the ability to process instructions. The software is a list of instructions that we create. The PIC will follow these to the letter with no exceptions, so we must make sure that they are written correctly or our program will not work as intended.

Define the problem

To begin planning we must first define the LED flash problem that is going to be solved by using a PIC. This is the physical needs of the project. You can't write reams of software without knowing what the PIC is going to control. You may find that you need to alter hardware and software as you progress, but don't be discouraged. This is normal for a lot of projects and is called 'Prototyping'.

We can start this discussion by saying that we must have a voltage source connected to the LED to make it light. Usually we put a resistor in series with the LED to limit the current through it to a safe level and in most LED's the maximum current is about 20mA.

Quite obviously, if the PIC is going to turn the LED on and off for us, then the LED must be connected to one of it's pins. These pins can be set as inputs or as outputs and when they are set as outputs we can make each individual pin have 5 volts connected or 0 volts connected by writing either a Logic 1 or a Logic 0 to them. We can now define this by saying we set a pin as an output high or as an output low.

When a pin is an output high it will have 5 volts connected to it and is able to source 20mA of current. When a pin is an output low it will have 0 volts connected to it and can sink 25mA of current.

A red LED will consume about 2 volts across it when it is being used. We know that an output pin will have 5 volts connected to it, so that means the series resistor needs to consume the remaining 3 volts. $5V - 2V = 3V$. By using Ohms law we can calculate the value of the resistor which must drop 3 volts with 3mA of current flowing through it and the LED.

$$V = I R \quad \text{or} \quad R = V / I \quad R = 3 / 0.003$$

Therefore $R = 1000$ ohms, or 1K.

Our circuit so far is a 1K ohm resistor in series with a red LED.

Which pin are we going to use to drive this LED? On the 16F84 there are 13 pins available for us to use and these are divided into 2 Ports.

PortA has 5 pins which are numbered RA0 - RA4.

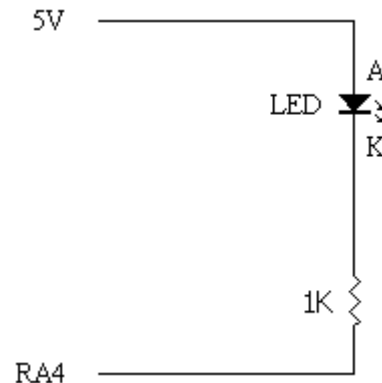
PortB has 8 pins which are numbered RB0 - RB7.

At this stage you might think that we can use any one of these, and you would be right - except for one thing. Pin RA4 is an **open collector**, which means that it can only connect a pin to 0 volts not 5 volts. Therefore our LED would not turn on if the LED was connected between this pin and ground. It would need to be connected between this pin and 5 volts.

There are lots of little hidden “gotcha’s” that exist in the world of microcontrollers and the best way of knowing about them is by close examination of the data book. You will remember most of these tricks after you become familiar with a particular chip, but even the most experienced programmers can get caught with these problems sometimes.

Even though we can use any pin to drive our LED circuit, for this example we will use the open collector pin RA4. We are going to use this pin because it highlights the fact that it operates differently from all the others.

To summarise our project so far, we are going to flash a red LED in series with a 1K ohm resistor from PortA pin RA4. RA4 can only sink current to ground, so the LED cathode is connected to this pin via a 1K ohm resistor. As you may already know, a LED can only work if the Cathode is more negative than the Anode, so the Anode side of the LED will be connected to the 5 volt supply.



You can easily tell the cathode from the anode. The cathode pin will have a flat surface moulded next to it into the red plastic, and the anode pin is longer than the cathode pin.

Now that we have our LED circuit figured out, the next stage is to write the software that will flash it for us.

To find out more about how the PIC Port Pins work, click on the blue link to run the program called [MicroPort](#). This is quite a lengthy program, so at this stage you may like to continue instead.

Look at the file called [program.pdf](#) to see how to build the Experimenters Kit. These files are located in the same directory where the software was installed. You can click on the blue links to activate them.

Writing The Software

There are quite a few ways to create software for the PIC. You can write it with a simple text editor, or use MPLAB from Microchip, a Basic compiler such as MicroBasic, or you can use the assembler called MicroPlan.

The PIC doesn't care what method you use to write the software because it only understands raw HEX code which is placed into the chip by using an appropriate programmer.

This is a sample of HEX code.

```
:1000000008308500003086008316173085003E30AA  
:10001000860083128F0190018221900B0C283330CF  
:1000200091003030920034309300413094005821D8  
:100030000F1817284C300E060319362856300E06B6
```

It's not very meaningful is it, but that does not matter to us. We are not computers, so it is not in our best interests to understand what this data means. What does interest us as programmers, is just how do we create a data listing like this.

The answer is - by using an Assembler.

This is quite an ingenious piece of software because it can read a text file that we have created for our program and turn it into a data file similar to the sample above. We can actually write our programs by collecting all the HEX values that our program will use, and then create a data file ourselves. The trouble with that idea is that it is a very tedious task and it would be terribly painful to try and find errors in it.

HEX numbers may be new to you so it will be best to have a quick look at them. At some stage, you may need to use them in your programs as well as Decimal and Binary numbers.

Number Systems

Any computer system, whether it be a PIC a PC or a gigantic main frame, can only understand these 2 things.

One's and Zero's - 1's and 0's.

The reason is quite simple. A computer is made up of millions of switches that can either be on or off. If a switch is on, it has a 1 state. If a switch is off, it has a 0 state. In computer terms these are called Logic States.

Logic 1 - switch is on.

Logic 0 - switch is off.

It is exactly as we mentioned before when we were talking about the output port pins of the PIC. If a pin is output high then it is Logic 1 or 5 volts. If the pin is output low then it is Logic 0 or 0 volts. If a pin is configured as an input and 5 volts is connected to this pin, then the PIC would 'see' a Logic 1. Similarly, the PIC would 'see' a Logic 0 on this pin if 0 volts were connected.

We were taught to count in a Base 10 or Decimal number system because we have 10 fingers on our hands. In this system, we add 1 to each number until we reach 9. We then have to add an extra digit to the number to equal ten. This pattern continues until we reach 99 and then we place another digit in, and so on.

The computer uses a Base 2 or Binary number system because it only has 2 Logic states to work with. If we have only got the numbers 1 and 0 to use, it would seem obvious that after the number 1, we have to start adding more numbers to the left. If we count from 0 to 15 this is how it would look in Decimal and Binary.

<u>Decimal</u>	<u>Binary</u>
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Now you may be able to see how a computer can represent numbers from 0 to 15. To do this it would need at least 4 switches because the number 15 represented in Binary is 4 digits long. Another way to say that, is the number is 4 *binary digits* long. The term “Binary Digits” is often abbreviated to “Bits”, hence our binary number is now 4 bits long, or just 4 bits. Perhaps we should redraw the table to emphasise the bit count. Remember, 0 bits represent a logic 0 value so we should not leave them out.

<u>Decimal</u>	<u>Binary</u>
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

The HEX number system is Base 16, that is you count 16 numbers before adding an extra digit to the left. This is illustrated in the table below.

<u>Decimal</u>	<u>Binary</u>	<u>HEX</u>
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Notice how letters are used after the number 9, and perhaps you can see why we only counted up to the number 15. This is a nice even 4 bit number and in computer terms this is called a NIBBLE.

If you look at the data sheet for the PIC 16F84, you will see that it is an 8 bit device. That means it can only deal with Binary numbers that are 8 bits wide. This is how 8 bit numbers are represented.

<u>Decimal</u>	<u>Binary</u>	<u>HEX</u>
0	00000000	00
1	00000001	01
2	00000010	02
3	00000011	03
4	00000100	04
5	00000101	05
6	00000110	06
7	00000111	07
8	00001000	08
9	00001001	09
10	00001010	0A
11	00001011	0B
12	00001100	0C
13	00001101	0D
14	00001110	0E
15	00001111	0F
16	00010000	10
17	00010001	11
18	00010010	12
- -	- -	- -
252	11111100	FC
253	11111101	FD
254	11111110	FE
255	11111111	FF

Looking at this table, you can see that an 8 bit Binary number can have a maximum value of 255 in Decimal or FF in HEX.

8 bit Binary numbers are called BYTES.

Our binary numbers can get quite large, and as they do so they will get more complicated and harder to understand. We are not computers, but we want to understand what we write for them, so with the help of an assembler we can use Decimal, Binary and Hex numbers in our software.

Have a look at a 32 bit binary number.

10001100101000001000110010100000

It's not hard to see why Hex and Decimal are easier to use. Imagine a 64 bit number and larger still. These are common in today's computers and are sometimes used with PIC's.

Lets try to see what this value equals.

First off, split the number into Bytes.

10001100 10100000 10001100 10100000

Looking Easier - Now split these into nibbles.

1000 1100 1010 0000 1000 1100 1010 0000

Even easier - Now convert these into Hex. This may be difficult at first, but persevere. After awhile you will be able to convert Binary to Hex and vice versa quite easily.

8 C A 0 8 C A 0

Combine the Hex numbers for our result.

8CA08CA0

To show that we are talking in Hex values we put in a little (h) after the number like this.

8CA08CA0h

Lets convert that hex number back to Decimal again.

8 C A 0 8 C A 0

Remember in Decimal, each value in the columns increases by a power of 10 as we move to the left, and in Binary they increase by a power of 2. In Hex, as you may have guessed, they increase by a power of 16. Be like me if you wish to, cheat and use a calculator to convert between the three number systems, it is much easier. In fact having a calculator that does these conversions is a very good investment for a programmer.

Lets start from right to left and convert each individual hex number to decimal.

Hex	Decimal	Multiplier	Calculate	Result
0	0	1	0 x 1	0
A	10	16	10 x 16	160
C	12	256	12 x 256	3,072
8	8	4,096	8 x 4,096	32,768
0	0	65,536	0 x 65,536	0
A	10	1,048,576	10 x 1,048,576	10,485,760
C	12	16,777,216	12 x 16,777,216	201,326,592
8	8	268,435,456	8 x 268,435,456	2,147,483,648

Add up the last column and we get a total of 2,359,332,000.

Therefore:

10001100101000001000110010100000 = 8CA08CA0h = 2,359,332,000.

Isn't it lucky that we don't have to think like computers.

The PIC can only deal with individual 8 bit numbers, but as your programming skills increase and depending on your software requirements, you will eventually need to know how to make it work with larger numbers.

When you combine 2 Bytes together, the Binary number becomes a WORD.

PROGRAMMING TIP:

Break large problems into many smaller ones because the overall problem will be much easier to solve. This is when planning becomes important.

Getting back to the topic of assemblers, you should remember that the program data file that we saw earlier is just a list of HEX numbers. Most of these numbers represent the program instructions and any data that these instructions need to work with.

To combine data and instructions together, the PIC uses a special Binary number that is 14 bits wide. If you look at the data sheet, you will see that the PIC 16F84 has 1024 14 bit words available for program storage. In computer language 1024 means 1K, so this PIC has 1K of program space.

We don't want to be overly concerned with Binary numbers when it comes to writing software so we need to become familiar with a language called ASSEMBLER. This language enables us to write code in such a way that we can understand and write it easily.

Some people do not want to write code in assembler because they think it is too hard to learn, and writing code in a language such as BASIC is much easier. Sometimes this is true, but sometimes you cannot write tight and fast code with these "higher level languages" and this may not be the best course of action for your particular project.

ASSEMBLER IS EASY - EASY - EASY.

This is especially so with PIC's because there are only 35 instructions to work with. Sometimes you will get into difficulty with assembler such as solving problems with multiply and divide, but these routines are freely available from many sources including the internet. Once you have them, it is usually only a simple matter of pasting them into your code if needed. Mostly, you will not even care how they work, but it is a good exercise to learn the techniques involved because it builds up your own individual knowledge.

Saving special code routines in a directory on your PC can be very productive and this is termed a 'Code Library'.

Mnemonics

This is a funny looking word. You pronounce it

Nem On Icks.

These are quite a powerful concept in programming because they provide an interface between us mere mortals and computers. It can become very confusing to write software if we have to refer to RAM addresses and data values by their binary numbers. Mnemonics makes it a lot easier for us to understand what we are writing by allowing us to assign names and labels to Instructions, RAM locations and Data values.

As an example, what do you think this means?

0000100000000011.

Any ideas?

What about this?

0803h

Try this.

movf 03h, 0

Lets change it to something we can understand using Mnemonics.

movf STATUS, W

That's a little easier to understand don't you think. It is exactly the same thing as the original Binary number except the first way the computer understands, the second and third ways we may understand, but the fourth way is quite easy to understand.

It means...

Move the contents of a file register called Status into W.

It's all too easy. Of course we still need to understand what MOVF, STATUS and W mean, but that will come soon.

The assembler is used to generate code that the PIC can understand by translating these mnemonics into binary values and store them as a HEX data file ready for a programmer to use. The standard assembler for a PIC is called MPASM for DOS, and MPASMWIN for Windows. These are free programs that are available from Microchip and are also supplied with **MICRO's** after you install MPLAB from the CD.

The Assembler

Writing code for an assembler is quite easy, but as with most things there are a few things to learn.

First off, the code you are writing for the assembler is called "Source Code". After the assembler has assembled the source code successfully, it generates another file that is called "Object Code". This is the HEX data file we saw earlier.

When you use the assembler program, there are some options that you can enable or disable such as, generating an error file, case sensitivity in your source code and a few others. Mostly you can ignore these and leave the default settings, but when you get more advanced at programming, you may like to change them to make the assembler work more suitable for your particular needs.

The assembler must be able to understand every line of source code that you write or it will generate an error. If this happens, the object code will not be created and you will not be able to program a chip.

Each line of code in the source file can contain some or all of these types of information. The order and position of these items on each line is also important.

- Labels
- Mnemonics
- Operands
- Comments

Labels must start in the left most column of the text editor. Mnemonics can start in column two and any other past this point. Operands come after the mnemonic and are usually separated by a space. Comments can be on any line usually after an operand, but they can also take up an entire line and are followed by the semi colon character (;).

One or more spaces must separate the labels, mnemonics and operands. Some operands are separated by a comma.

Here is a sample of the text that an assembler expects.

```

        Title    "mICro's Simple Program"

        list p=16f84      ; processor type

;
; -----
; PROGRAM START
; -----
;
        org 0h            ; startup address = 0000

start    movlw 0x00        ; simple code
        movwf 0x05
        goto start        ; do this loop forever

        end
```

You can see the label called `start` in column 1. Then comes a tab or space(s) to separate the label from the mnemonic `movlw`. Then comes another tab or space(s) to separate the mnemonic from the operand `0x00`. Finally, you can see the comment which comes after the semi colon `; simple code`

Lets have a look at his a bit closer.

The first line has a `Title` directive. This is a predefined part of the assemblers internal language and lets you define a name for your source code listing. The next line has a `list` assembler directive and tells the assembler what type of processor it is assembling for. In this case, a 16F84 chip. This can be helpful if you write too much code for this particular processor to use. The assembler will decide this while it is building the object code and generate an error if it encounters a problem. Directives like these control how the assembler builds the final Object code, and there are quite a few of them available for use so it would be best to consult the help file for MPLAB or MPASM for further details.

The next lines are just comments put there to help you the programmer know what is going on.

It is vital that you get into the habit of writing comments all over your code. This will help you understand what you have written especially when you come back to read it another time. Believe me, it is very easy to get confused trying to follow the meaning of your code if there is no explanation on how it works.

The next lines tell the assembler the meanings of user defined labels.

`org 0h` is another directive that tells the assembler to set the current ROM address where the following instructions will be placed from. Remember that the 16F84 has 1024 location available for code use. `org 0h` tells the assembler to start loading the instructions starting from ROM address 0. In other words the `movlw 0x00` instruction will occupy ROM location 0 and the `movwf 0x05` instruction will occupy ROM location 1. `goto start` will occupy ROM location 2.

One thing you may have missed here, is the fact that ROM and RAM addresses start from location 0, not location 1. The ROM space is actually from 0 to 1023, not from 1 to 1024. Remember 0 is a valid Binary number.

It is probably best to use the correct terminology now to refer to ROM locations as ROM addresses.

`goto start` is an instruction mnemonic followed by a label name. The assembler knows that the instruction `start movlw 0x00` is at ROM address 0, so when it sees the instruction `goto start`, it will generate code to tell the PIC's processor to goto ROM address 0 to fetch the next instruction.

So what happens if there is no label called `start` anywhere in the source listing? The assembler will complain that it cannot find the label, generate an error and will not complete the assembly process. Usually these errors are written to a separate error file and in a listing file as well. These files have the same name as your source file but with `*.err` and `*.lst` extensions.

You cannot have two labels like this with the same name either. That confuses the assembler because it does not know which particular line you are referring to. This also generates an error.

Don't get too worried about error messages. They are simply there to help you find problems with the way you wrote your code. Look at the list file that is generated to see the offending line, fix the problem back in the source code and then reassemble.

You will also receive "warning messages" at times. These are generated to tell you that you may be doing something wrong, like forgetting to set a page bit.

Warning [205]: Ensure page bits are set.

If you are certain that your code is correct, you can ignore them. We will talk about page bits later.

```
start    movlw 0x00                ; simple code
          movwf 0x05
          goto start                ; do this loop forever
```

Code that is written like this is called a LOOP. That is because the code executes until the `goto start` instruction forces the processor to begin again. This particular code will LOOP forever or until you remove power from the chip. Your code must have some sort of loop to keep it flowing.

```
start    movlw 0x00                ; simple code
          movwf 0x05
```

If you had code like this, what would happen after the `movwf 0x05` instruction was executed? The answer is that the PIC would get lost because you have not given the processor anything to do past this point.

So what exactly does this small piece of code do?

Remember our LED flash problem?

The LED was connected to pin RA4 which is PortA pin 4.

PortA has 5 pins available for us to use, but we were only going to use pin RA4 and forget about the others. If we had to turn the LED on we have to write a Logic 0 to this pin. Remember that once we have done this, 0 volts will appear on the pin if it is set as an output.

What value should we write to Port A if we want to set pin RA4 to Logic 0? Here is a small sample of a binary table.

<u>Decimal</u>	<u>Binary</u>
0	000000
1	000001
2	000010
3	000011
16	010000
47	101111

If we consider that pin RA4 is represented by bit 4 in this table we can see that we can write any value as long as this bit equals 0.

Another thing you must consider here, is that we can do this because nothing else is connected to Port A. If some devices were connected to the other Port A pins, we need to be more specific about the value we write to Port A so that we do not upset their operation.

In our case, to turn the LED on, we can write the value 0 to Port A, and to turn the LED off we can write 16 to PortA. So just how do we do this? If you look at the original code, this is the first line.

```
start      movlw 0x00                ; simple code
```

First we have a label called `start`, followed by the instruction `movlw 0x00`.

`movlw` means to **move** a **literal** value into the **W** register. A literal value is any value that can fit into 8 bits. Remember that the PIC is an 8 bit device. This means a literal value can be any value from 0 to 255. In this case it is the value 0, or 0x00 which is just a way of writing HEX values. This instruction can also be written as

```
movlw b'00000000'      ; binary notation
movlw d'0'              ; decimal notation
movlw 0h                ; another type of HEX notation.
```

Binary notation is quite good for writing to the Ports because any bits that are 1 means the corresponding output Port pin will be at 5 volts, and any that are 0 will be at 0 volts. The exception is pin RA4 which is at a high impedance state when it is at Logic 1.

See the program called [MicroPort](#) for more details.

Notice the two styles of HEX notations. If any HEX number begins with a letter it must use the '0x' notation. This is a zero and x. Any other HEX numbers can be written in either notation.

```
movlw 0xAA      movlw 34h      movlw 2h
movlw 0x33      movlw 0x00      movlw 0xEA

movlw FFh      This is not allowed, it must be:  movlw 0xFF
```

Looking back at the code, the next line is

```
movwf 0x05
```

`movwf` means to **move** the contents of the **W** register to the file register specified. In this case it is RAM address `0x05`.

If you look at the 16F84 data sheet, and most other PICs for that matter, you will see that RAM address 5 is Port A. So in other words, this instruction moves the contents of W to Port A.

This seems a lot of effort. Why can't we just write 0 to Port A? Unfortunately the PIC does not function like that. You cannot directly write any 8 bit values to any RAM locations. You have to use the W register to do the transfer. That is why it is called the W or 'Working Register'.

Labels

We mentioned the use of labels before. With an assembler, we have the luxury of being able to create our own label names which we can use to define things like general RAM addresses, special RAM locations, port pins and more.

As an example of this concept we can change our original code...

```
start      movlw 0x00                ; simple code
            movwf 0x05
```

into this...

```
start      movlw TurnOnLED           ; simple code
            movwf PortA
```

By writing your code in this way, you can just about comprehend the meaning of these two code lines.

Write a special value to PortA which turns on a LED.

Now this is all very fine except for one thing. How does the assembler know the meaning of `TurnOnLED` and `PortA`? It has the ability to understand all the PIC instructions like `movlw`, and also what labels are, but you, as the programmer, have to tell the assembler the meaning of any 'new' labels that you create.

To do this you use the `equ` assembler directive. This tells the assembler that the label on the left of the `equ` directive has the value on the right side.

```
TurnOnLed   equ 0x00                ; value to turn on LED with RA4
PortA       equ 0x05                ; PortA RAM address
```

You should note something here, and that is the first equate assigns a value to a Label that will be used as a Literal, and the second equate assigns a value to a Label that will be used as a RAM address. These are quite interchangeable by the way because they are just simple numerical values.

```
start      movlw PortA           ; simple code
           movwf TurnOnLED
           goto start           ; do this loop forever
```

This new piece of code is quite valid and makes sense to the assembler. When the PIC executes this code it will now get the Literal value 0x05 and place it in W, and then it will get this value from W and place it into RAM address 0x00. This address is where indirect addressing takes place. Check this out for yourself in the data book. This code does not make much sense now, but the assembler does not care what we write. It's only concern is that it can successfully assemble this code.

Now that we know about Labels, this is how we can rewrite the original code listing and make it more readable

```
                Title  "mICro's Simple Program"

                list p=16f84                ; processor type
;
; -----
; PROGRAM START
; -----
;
TurnOnLed equ 0x00        ; value to turn on LED with RA4
PortA     equ 0x05        ; PortA RAM address

                org 0h                ; startup address = 0000

start      movlw TurnOnLed           ; simple code
           movwf PortA
           goto start           ; do this loop forever

           end
```

Quite simple isn't it.

One thing to note is that label names must start in the first column on a separate line, and you cannot have spaces or TABs before them.

Now each time the assembler comes across a Label called `TurnOnLED` it will substitute it for the value `0x00`. When it comes across a Label called `PortA` it will substitute it for the value `0x05`. The assembler does not care in the least what these labels mean. They are there just to make it easier for us to read our code. Lets have a quick look at how the assembler turns the source code into a HEX file.

When the assembler begins working, it creates a symbol table which lists all the labels you created and then links the values you associated with them.

```
TurnOnLed equ 0x00      ; value to turn on LED with RA4
PortA      equ 0x05      ; PortA RAM address
```

The symbol table will look like this.

SYMBOL TABLE	
LABEL	VALUE
PortA	00000005
TurnOnLed	00000000

The assembler also has a ROM address counter which is incremented by 1 each time it assembles a line of code with an instruction in it. This next line is called an assembler directive and it tells the assembler to set this counter to ROM address 0.

```
org 0h                      ; startup address = 0000
```

The next code line has an address label attached to it so the assembler also adds this to it's symbol table. At this stage the ROM address counter equals 0 so the `start` label gets the value 0.

```
start      movlw TurnOnLed      ; simple code
```

The symbol table will look like this.

SYMBOL TABLE	
LABEL	VALUE
PortA	00000005
TurnOnLed	00000000
start	00000000

Next on this code line is `movlw`. If you look at the data book the `MOVLW` instruction has a Binary number associated with it. Remember about computers only understanding 1's and 0's. This is how the PIC understands instructions.

MOVLW in Binary = 11 00XX kkkk kkkk

We need to decipher this a bit.

The 1100XX is the part of the instruction that tells the processor that it is a MOVLW instruction. The 'XX' part means that it doesn't matter what value these two bits are. The 'kkkk kkkk' represents the 8 bits of data and will be the actual Literal Value. The assembler will look up it's symbol table and find the label TurnOnLED and return with the value 0. It will then insert this information into the MOVLW instruction data.

Therefore the complete instruction becomes 11 0000 0000 0000 which is 3000h.

Notice there are 14 bits for the instruction, and that the instruction itself is represented with the literal data combined. In this way each PIC instruction only occupies 1 single ROM address in the chip. Therefore the 16F84 with 1K of ROM space, can store 1024 instructions in total.

The assembler is now finished with this code line because it does not care about the comment, so it increments it's address counter by 1 and continues with the next line.

```
movwf PortA
```

MOVWF = 00 0000 1fff ffff

00 0000 1 are the bits that define the MOVWF instruction and fff ffff are the bits that define the RAM address where the W register contents will end up. The assembler looks up PortA in it's symbol table and returns the value 5.

Therefore the complete instruction becomes 00 0000 1000 0101 which is 0085h.

So far the assembler has generated the HEX numbers 3000h and 0085h.

This is the next line.

```
goto start ; do this loop forever
```

GOTO = 10 1kkk kkkk kkkk

10 1 are the bits that define the GOTO instruction and the kkk kkkk kkkk are the 11 bits that define the new ROM address that the processor will begin from after this instruction executes.

If you do the math you will see that a number with 11 bits can range from 0 to 2047. This means that a `GOTO` instruction has enough information to let the processor jump to any ROM address from 0 to 2047. The 16F84 only has a ROM address space that ranges from 0 to 1023 so a `goto` instruction can let the processor jump to anywhere in this device.

The assembler will look up the symbol table for the label name `'start'` and return with the value 0.

Therefore the complete instruction becomes `10 1000 0000 0000` which is `2800h`.

The final HEX code assembled for this little program is `3000h 0085h 2800h`.

This next line is another assembler directive and tells the assembler that this is the last line of the source file. Please don't leave it out or the assembler will get upset and generate an error.

`end`

It's not hard to imagine now why the assembler will complain if you leave out a label name or use one that was spelled incorrectly.

After assembly the next page shows what the listing file will look like. If the name of your source file was `first.asm` then the list file will be `first.lst` and the HEX file will be `first.hex`. These new files will be created in the same directory where `first.asm` is located and are simple text files which can be viewed with any text editor program such as 'Notepad'.

If you look through this listing you should be able to verify what has just been mentioned.

On the far right of the listing, are the current ROM addresses and these increment as each instruction is added. You will notice that they start off at 0 because of the `ORG 0h` directive.

If a code line has an instruction on it, then next to the ROM address, you will see the 4 digit HEX code that represents the instruction.

After the `end` directive you will see the Symbol Table contents, then how much of the processors memory was used, and finally, some information on errors that the assembler may have encountered.

LOC	OBJECT CODE	LINE	SOURCE TEXT
VALUE			
	00001		Title "mICro's Simple Program"
	00002		
	00003		list p=16f84 ; processor type
	00004		
	00005		; Copyright Bubble Software 2000
	00006		;
	00007		; -----
	00008		; PROGRAM START
	00009		; -----
	00010		;
00000001	00011	TurnOnLed	equ 0x00 ; value to turn on LED with RA4
00000005	00012	PortA	equ 0x05 ; PortA RAM address
	00013		
0000	00014		org 0h ; startup address = 0000
	00015		
0000 3000	00016	start	movlw TurnOnLed ; simple code
0001 0085	00017		movwf PortA
0002 2800	00018		goto start ; do this loop forever
	00019		
	00020		
	00021		end

MPASM 02.30 Released
PAGE 2
mICro's Simple Program

FIRST.ASM 12-31-1999 17:26:38

SYMBOL TABLE

LABEL	VALUE
PortA	00000005
TurnOnLed	00000000
__16F84	00000001
start	00000000

MEMORY USAGE MAP ('X' = Used, '-' = Unused)

0000 : XXX-----
--

All other memory blocks unused.

Program Memory Words Used: 3
Program Memory Words Free: 1021

Errors : 0
Warnings : 0 reported, 0 suppressed
Messages : 0 reported, 0 suppressed

After successful assembly, this is what the generated HEX file looks like.

```
:060000000030850000281D
:00000001FF
```

This has all the information that a programming device needs to write your program code to a chip. The default HEX file generated by MPASM is a style called INHX8M, and this is how it is dissected.

```
:BBAAAATTLLHH....LLHHCC
```

BB This is a 2 digit value of the number of bytes on the line. 16 MAX

AAAA The starting address of this line of data.

TT A record type. Normally is 00, but will be 01 on the last line.

LLHH A data word presented as low byte high byte format.

CC The checksum value of this line of data.

```
:060000000030850000281D
```

:06 means 6 bytes of data on the line.

Our new code was 3 WORDS long which = 6 BYTES.

```
:060000000030850000281D
```

0000 means the bytes on this line are programmed starting from ROM address 0

```
:060000000030850000281D
```

00 means record type, but not on last line

```
:060000000030850000281D
```

00 30 85 00 00 28 are the 6 data bytes in low byte/ high byte format.
If you swap the bytes around and merge them into 3 words, you get

3000 0085 2800

This is the same as our code data.

:060000000030850000281D

1D is the checksum for all the data listed on this line. It is sometimes used by a programmer to make sure the data on each HEX code line has not been corrupted.

:00000001FF

The last HEX line has 0 bytes listed and 01 in the record type which means it is the last line of the file.

Using a text assembler

If you haven't installed MPLAB on your computer yet, please do so now. It is located in a separate directory on the **MICRO's** CD. Wait for the auto start when you insert the CD into the drive and click on 'Install MPLAB' when the intro screen is shown. When this program is installed you will have access to the MPLAB Integrated Development Environment by Microchip, and also makes MPASMWIN available for separate use.

For convenience, create a MPASMWIN shortcut for your desktop. Right click on a blank part of the desktop screen and a pop up window will appear. Click on 'New - Shortcut' and a dialog box will appear. Click on the browse button to display a directory dialog box. Double click on 'Program Files - Mplab - Mpasmwin.exe'. Now click on 'Next' and change Mpasmwin.exe just to Mpasmwin in the edit box displayed. Click finish and a new Mpasmwin icon will appear on the desktop.

Now we are going to do our first assembly using this program.

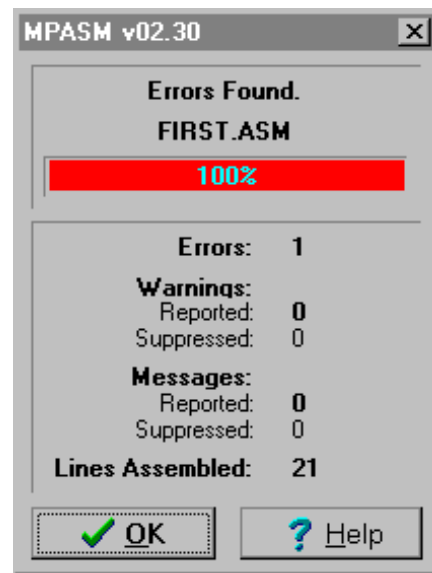
Double click on this new icon to start Mpasmwin. You will notice the screen shows various options which are the assembler directives that were mentioned earlier. At this stage you do not need to worry about them as we will use the default ones shown.

Click on the Browse button to show a directory dialog box. Double click on the 'c:\' icon to bring the directory listing back to the C drive root directory. Now look down this directory to find the softwareinstallation directory. The default name will be 'Bubble'. In the directory list on the left side you should be able to find a file called `first.asm`. This is the source code for the file we have just been looking at. Double click on this and it will be listed in Mpasmwin as the file it will assemble. Now click on 'Assemble' and Mpasmwin will assemble this file.

If all went well, a new dialog box will be displayed with a green progress bar saying that assembly was successful. If all didn't go well, this bar will be red and you will have to find out where the problem is in the source code before continuing.

I'll just bet that the assembly process failed and that bar is a bright red colour, and the screen tells us that there was 1 error encountered. Guess what? Now we have to fix the problem.

First, click on the OK button to close Mpasmwin. Start the program called 'Notepad' which is located in 'Start - Programs - Accessories - NotePad'. Perhaps you can create a short cut for this program as well. Now click on 'File - Open' and open the file called `first.asm` which is located in the software installation directory.



See if you can see where the problem is.

If you can't then don't worry, Mpasmwin will tell us. Using Notepad, open up the file called 'first.err', and this should be the contents.

```
Error[113] C:\...\FIRST.ASM 16 : Symbol not previously defined (TurnOnLed)
```

It is telling us that a Symbol has not been previously defined, notably, `TurnOnLed`. You can see that the error occurred on line number 16 of the source file. To make things easier, use Notepad to open the file called `first.lst`. This will look very similar to the listing shown previously. The only difference between them is an error listed.

```
Error[113] : Symbol not previously defined (TurnOnLed)
0000 3000          00016 start      movlw TurnOnLed          ; simple code
```

The offending line is the one following the error message. Now that you can actually see the problem, use Notepad to reopen the source file `first.asm`.

Now this seems crazy. We have defined `TurnOnLed` and we have used it in a way that should make sense to the assembler.

Look a little closer and you may see the problem.

Here are the two lines that define and use the label.

```
TurnonLed equ 0x00          ; value to turn on LED with RA4  
start      movlw TurnOnLed    ; simple code
```

Can you spot the problem yet?

Is TurnonLed the same as TurnOnLed?

No it isn't, because TurnonLed has a lowercase 'o' and TurnOnLed has an uppercase 'O'. This makes a big difference to the assembler. Start Mpasmwin again and you will notice a 'Case Sensitive' checkbox which is checked. This means the assembler treats 'o' as a different character to 'O'. You can decide if you want case sensitivity or not when you write your programs.

You might think that the offending line should be the one that defined TurnonLed, but this is incorrect. If you look at the listing file again, you will see that the assembler has TurnonLed listed in the symbol table.

```
TurnonLed          00000000
```

Then when it got to the line with TurnOnLed, it could not find this label in the symbol table, so it told us of the error at this point.

Use NotePad to open `first.asm` again and change the TurnonLed symbol to TurnOnLed and save the file.

```
TurnOnLed equ 0x00          ; value to turn on LED with RA4
```

Start Mpasmwin again and `first.asm` should still be listed for assembly. Click on 'Assemble' and this time you should have a green bar highlighted showing success.

There are lots of different errors that may appear from time to time, and part of the art of programming is developing some expertise in finding and fixing them.

Sometimes the assembler will not generate an error, but the code still doesn't do what we wanted. This means that we made an error with the way we wrote the code and at times this can be very hard to find. Good documentation can ease the pain so make sure you have enough listed to make you understand your code as much as possible.

Use Notepad to open the file called `first.hex` which has been created in the software directory by Mpasmwin. It should match up with this which will be the same as the HEX listing that was shown previously.

```
:060000000030850000281D  
:00000001FF
```

OK, OK, I know. I said we will be creating a LED flasher as our first program, but if you think you have got a good grasp on what has been said up to this point, then please feel free to move on. If you are not sure then it is best to go back and have another look.

The Flowchart

The flowchart is another part of the planning stage for our projects and they are very good for documenting larger programs as they enable us to see a pictorial representation of the flow of our programs. To create a flow chart, you just need to figure out what your program will do. If it is a large program, don't try to cram everything into a huge flow chart. Break things down into simpler tasks.

You might think that the previous program will turn on the LED if we programmed a PIC and connected a clock source and applied power. Unfortunately you would be mistaken. The LED will not come on at all.

The reason is because we have not told the PIC to make Port A pin RA4 work as an output. If you look at the 16F84 data sheet, you will discover that Port pins are set as inputs as a default setting, so there are no means yet to drive the LED.

If you think about it, this should be our first course of action - to set the port pins appropriately so that we can get control of external devices connected to the PIC as quickly as possible after we turn the power on.

After we have done that, our next task is to make the LED turn on and off. Do we make the LED flash fast, medium or slow? These are things you have to decide on. Lets say we will make it flash every half second, which is 500 milli seconds, or 500mS. The LED will be on for 500mS then off for 500mS.

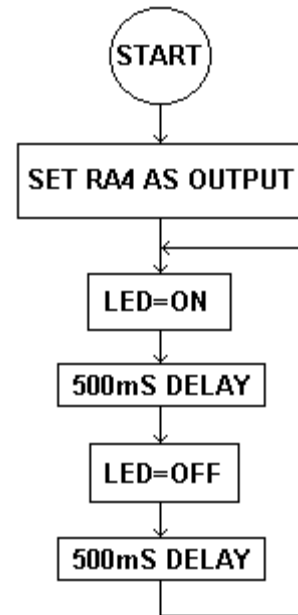
Somehow we need to make Port A pin RA4 Logic 1 for 500ms and then Logic 0 for 500mS. The 500mS is called a 'DELAY' and this must be used each time the Logic state of RA4 is changed.

To keep the LED flashing continuously we need to create a 'LOOP' in our code otherwise the LED would flash just once and stop.

This then will be the basis for our flowchart.

Flowcharts are easy to create. They are usually just text surrounded by boxes, diamonds and circles which are connected by lines with arrows showing the direction of program flow. Here is the flow chart for the LED flash program. As you can see it is quite simple and shows in a pictorial fashion what we mentioned on the previous page.

They usually begin with a START statement inside a circle. If you follow the connected line downwards in the direction of the arrow the next box tells us to set Port pin RA4 as an output. Next comes the box to turn the LED on followed by a 500mS delay. Then we turn the LED off followed by another 500mS delay. After that, the line LOOPS back to turn the LED on again and this will continue while power is applied to the chip.



When you look at the program with a flowchart it is quite easy to follow what is going on. You can write whatever you like in these boxes, but just make sure you understand what is going on when it is finished. Create them in pencil to start with because you will probably change things a quite bit while you develop your projects. Now we are ready to begin writing the flash program. We can use most of what we have seen already to start with.

```

Title    "mICro's Flash Program"

        list p=16f84      ; processor type
TurnOnLed equ 0x00        ; value to turn on LED with RA4
PortA     equ 0x05        ; PortA RAM address
;
; -----
; PROGRAM START
; -----
;

        org 0h            ; startup address = 0000

start    movlw TurnOnLed   ; simple code
         movwf PortA
         goto start        ; do this loop forever

end
  
```

All that's been changed so far is the `Title`. The first function block in the flow chart tells us to setup pin RA4 as an output. This is going to require 3 lines of code to do because we need to set a special RAM location that requires a programming trick to access. This is common to most PICs and is called selecting a RAM Page.

RAM addressing can get a little bit complicated inside the PIC, but once you know what is going on it is easy to understand. This topic is covered in detail in the [MicroPost](#) tutorial program.

We will look at this briefly here. Remember the MOVWF instruction and how it has 7 bits available in the instruction that specifies a RAM address.

```
MOVWF = 00 0000 1fff ffff
```

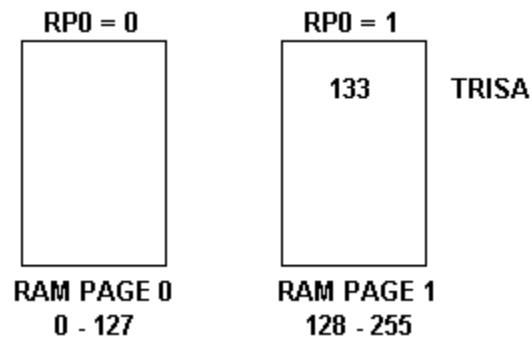
If you do the Binary math with 7 bits you will see that the maximum number that can be represented is 127 decimal. (111 1111 or 7Fh) This means that the MOVWF instruction can only move data to RAM addresses between 0 and 127.

The register that controls how we make the Port pins behave as an Input or an Output is called the TRIS register. There are 2 of these in the 16F84, one for Port A and is called TRISA, and another for Port B and is called TRISB.

The RAM address for these registers are 133dec and 134dec which is 85h and 86h respectively. So how can we use the MOVWF instruction to move data into the TRISA register to set pin RA4 as an output instead of being the default input when we only have 7 bits available in the instruction?

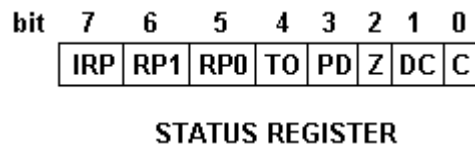
The answer lies in what is called a 'RAM Page Select Bit'. There are 2 RAM pages available in the 16F84. Page 0 has RAM addresses ranging from 0 - 127, and Page 1 has RAM addresses ranging from 128 - 255. Addresses 128 - 255 need 8 bits of information which is one more than can be used with the MOVWF instruction. So to use this instruction with the TRISA register we need to get the 8th bit from somewhere else.

This bit comes from the STATUS register, which is at RAM address 3, and is called the RP0 bit. Each time you use the MOVWF instruction, RP0 is used as the 8th address bit. If this bit is set to Logic 0 then the MOVWF instruction can access RAM addresses from 0 - 127. If this bit is set to Logic 1 then the MOVWF instruction can access RAM addresses from 128 - 255.



RP0 is bit number 5 in the STATUS register.

Notice how the bit numbers start from 0 on the right side and finish with 7 on the left side.



If we want to set the RP0 bit to logic 1 so that we can send data to the TRISA register, we can use this instruction.

```
BSF 03h,05h ; set RP0 for RAM page 1
```

What looks easier to read, the instruction or the comment?

Either you say!

What is BSF?

BSF means Bit Set File, or more simply, set a particular bit in a RAM register to Logic 1. The instruction as written means to set a bit number 5 in RAM register 3.

That looks a terrible way to write instructions, so what we want to do is to rewrite this instruction so that it makes more sense to us.

As you are now aware, we can do this by defining new labels. It's just a simple matter to add these new label definitions to our existing program.

```

                Title    "mICro's Flash Program"

                list p=16f84    ; processor type
TurnOnLed equ 0x00    ; value to turn on LED with RA4
PortA     equ 0x05    ; PortA RAM address
Status    equ 0x03    ; Status RAM address
RP0       equ 0x05    ; Status RP0 bit = bit 5
;
; -----
; PROGRAM START
; -----
;

                org 0h                ; startup address = 0000

start        movlw TurnOnLed        ; simple code
              movwf PortA
              goto start            ; do this loop forever

              end

```

Now that we have done that, we can use the labels in our program to set the Status RP0 bit so that we can access TRISA.

So instead of writing

```
bsf 03h,05h    ; set RP0 for RAM page 1
```

we can now write

```
bsf Status,RP0    ; set RP0 for RAM page 1
```

The opposite to this instruction is BCF which means to Bit Clear File, or more simply, clear a particular bit in a RAM register to Logic 0. After we set the TRIS register we need to set RP0 to Logic 0 so that we can access the PortA register which, if you remember, is RAM address 5 and therefore is in RAM Page 0.

We can write that instruction as

```
bcf Status,RP0    ; set RP0 for RAM page 0
```

In between these two instructions we need to set the TRISA register so that pin RA4 becomes an output. Do you remember how each bit in the PortA and PortB registers corresponds to the Port pins? Bit 0 in PortA = RA0, Bit 1 = RA1 etc.

The same is for the TRIS registers. Bit 0 in TRISA controls whether RA0 is an input or an output, Bit 1 for RA1, Bit 2 for RA2 etc.

To make any pin an output we clear the corresponding TRIS bit to Logic 0, and to set any pin as an input we set the corresponding TRIS bit to Logic 1.

This is easy to remember.

1 = Input
0 = Output

The default state of the port pins on power up are inputs, which means all the bits in both TRIS registers are set to Logic 1.

Now, to make Port pin RA4 be an output we need to set TRISA bit 4 to Logic 0.

These are the 3 instructions to do this task.

```
bsf Status,RP0      ; set RP0 for RAM page 1
bcf TrisA,RA4        ; set RA4 = output
bcf Status,RP0      ; set RP0 for RAM page 0
```

Before this will work with the assembler, we need to add the Labels TRISA and RA4 to our source code.

```
        Title  "mICro's Flash Program"

        list p=16f84      ; processor type
TurnOnLed equ 0x00        ; value to turn on LED with RA4
PortA     equ 0x05        ; PortA RAM address
TrisA     equ 0x85        ; TRISA RAM address
RA4       equ 0x04        ; PortA RA4 = bit 4
Status    equ 0x03        ; Status RAM address
RP0       equ 0x05        ; Status RP0 bit = bit 5
;
; -----
; PROGRAM START
; -----
;

        org 0h            ; startup address = 0000

        bsf Status,RP0    ; RP0 set for RAM page 1
        bcf TrisA,RA4     ; set RA4 = output
        bcf Status,RP0    ; RP0 set for RAM page 0

start    movlw TurnOnLed   ; turn on LED on RA4
        movwf PortA
        goto start        ; do this loop forever

        end
```

We have also added the code to make RA4 an output before the program does anything else. That way we gain control of the Port pins very quickly.

The next thing to do according to our flowchart is to turn the LED on. This is already in the code, so the next thing to do is create some code to make a 500mS delay.

```
start      movlw TurnOnLed           ; turn on LED on RA4
           movwf PortA
```

To make a simple delay, we need to make the processor waste the time required for the delay and an easy way to do this is to subtract 1 from a RAM register until it equals 0. It takes the PIC a certain amount of time to do this, but the trick is to find out how much time.

The processor used in the MiniPro PCB runs with a clock speed of 4MHz. That is 4 million cycles per second.

The PIC needs 4 of these cycles to process most instructions, which means they execute at a rate of 1 million per second. These are called Instruction Cycles.

This means that 1 instruction cycle = 4 clock cycles.

Some instructions, like GOTO and CALL, use 2 instruction cycles to complete.

For our purposes, all we have to do is figure out how many instruction cycles we need to waste to create a 500mS delay. Each basic instruction cycle with a clock speed of 4MHz takes 1 micro second to execute, so we need 500,000 of them to make our delay.

Let's create a small loop that simply decrements a RAM register until it equals 0.

```
WaitHere   clrf DelayL               ; clear DelayL to 0
           decfsz DelayL,f           ; subtract 1 from DelayL
           goto WaitHere             ; if not 0, goto WaitHere
```

Inside the 16F84, there are 68 general purpose RAM registers that we can use for whatever we need. The first of these registers starts at RAM address 12dec, or 0x0C. We need to use one of these to create the delay code loop above and we have called it `DelayL`. We can define the label like this so that we can use it in our program. This means the newly define RAM location is at RAM address 20 hex, or 32 decimal.

```
DelayL      equ 0x20      ; delay register LOW byte
```

The `CLRF DelayL` instruction makes the contents of RAM register `DelayL` equal zero, and gives us a known value to start from.

The `DECFSZ DelayL,f` instruction means to decrement the contents of `DelayL` and if it now equals 0 then skip over the next instruction.

Notice the `,f` that follows `DelayL`. This is called a Destination Designator. If the letter `f` is used, then the result of the operation is placed back into the register specified in the instruction. If the letter `w` is used then the result of the operation is placed into the W register.

Sequence of events for `DECFSZ DelayL,f`

- W Register = 0xA0
- DelayL = 0
- Value is read into the Arithmetic Logic Unit (ALU) = 0
- Value in ALU is then decremented = 0xFF
- DelayL = 0xFF
- W Register = 0xA0

Sequence of events for `DECFSZ DelayL,w`

- W Register = 0xA0
- DelayL = 0
- Value is read into the Arithmetic Logic Unit (ALU) = 0
- Value in ALU is then decremented = 0xFF
- DelayL = 0
- W Register = 0xFF

There are quite a few instructions that use `,f` or `,w` after the instruction to specify where the result goes.

The assembler doesn't care if you omit to use `,f` after an instruction, as it will assume that you want the result placed back into the specified register.

The assembler will generate a warning message like this, but you can ignore it as long as you know your code is correct.

```
Message[305]: Using default destination of 1 (file).
```

Notice the 1 value. As with all labels used with an assembler, they must have a value assigned to them. `f` and `w` are labels as well, but we don't need to worry about them as they are automatically assigned a value by the assembler.

```
f      assembler assigns value 1
w      assembler assigns value 0
```

So how does this value fit inside an instruction.

Remember the MOVWF instruction?

```
MOVWF = 00 0000 1fff ffff
```

Can you see the bit that equals 1? This is the destination bit which forms part of this instruction. The processor checks this bit during execution and if it is 1, sends the result back the specified RAM address. MOVWF always sends the result back to the specified RAM address, that is why this bit is always 1.

Instructions like this next one are a little different. DelayL = RAM address 0Ch.

```
DECFSZ DelayL,w
```

The binary code for this instruction is 00 1011 0000 1100, and the 'd' bit = 0.

```
DECFSZ DelayL,f or DECFSZ DelayL
```

The binary code for this instruction is 00 1011 1000 1100, and the 'd' bit = 1.

In future we will not use `,f` when we want the result of an instruction placed back into the specified RAM address. Now we can move back to the code listing.

When this code block executes, `DelayL` will be set to 0. Then it will be decremented by one and it will have a value of 0xFF. This new value is not equal to 0, so the next code line is NOT skipped. Therefore the instruction `Goto WaitHere` is executed and the processor loops back to the code line with the label `WaitHere`. `DelayL` is decremented again, and eventually it will equal 0 after this code has completed 256 loops. When this happens, the instruction `goto WaitHere` will be skipped thus breaking the loop and ending the delay.

The `DECFSZ` instruction takes one instruction cycle to complete unless the result of the decrement equals 0. It will then take two instruction cycles. A `GOTO` instruction always takes two instruction cycles to complete. If you do the math, it will take around 768 instruction cycles to complete this routine. This is quite a bit shorter than the 500,000 we need and if we were to use the delay routine as it is, the LED would flash on and off so fast, we would not see it.

If you would like to know more about instruction timing there are animated tutorials available in the program called [MicroPrac](#).

What we need to do is use this same delay routine enough times so that 500mS is wasted and we can accomplish this by using what is called a 'Nested Loop'. These are just code loops within code loops and to create one we use another RAM register to control how many times the existing delay code executes. If this is still not enough for the delay we need, then we will have to use more nested loops and more RAM registers.

In fact, for a delay of 500mS we need to use 3 RAM registers when the chip is executing instructions at a rate of 1 million per second.

Let's define these registers so we can use them.

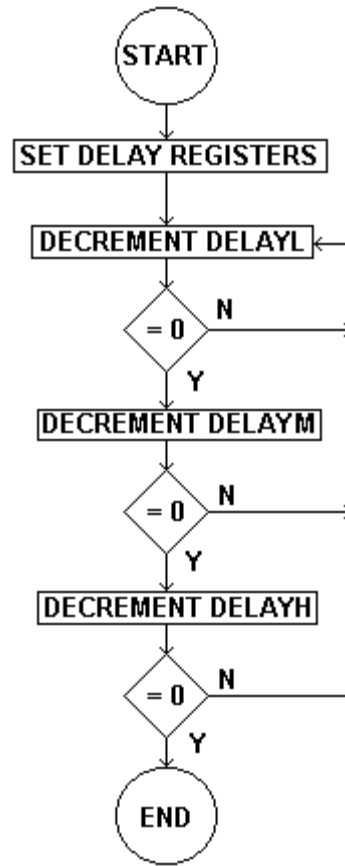
```
DelayL    equ 0x20        ; delay register LOW byte
DelayM    equ 0x21        ; delay register MID byte
DelayH    equ 0x22        ; delay register HIGH byte
```

Now we can construct a delay routine using these RAM locations with 3 nested loops.

```
                clrfs DelayL                ; clear DelayL to 0
                clrfs DelayM                ; clear DelayM to 0
                movlw 3h                    ; set DelayH to 3
                movwf DelayH
WaitHere    decfsz DelayL                    ; subtract 1 from DelayL
                goto WaitHere                ; if not 0, goto WaitHere
                decfsz DelayM                ; subtract 1 from DelayM
                goto WaitHere                ; if not 0, goto WaitHere
                decfsz DelayH                ; subtract 1 from DelayH
                goto WaitHere                ; if not 0, goto WaitHere
```

In this routine DelayL will get decremented until it equals 0 as mentioned before. Then DelayM gets decremented, because the first goto WaitHere instruction gets skipped. DelayM will now equal 0xFF, so the processor executes the second goto WaitHere and starts decrementing DelayL again. This double loop will continue until DelayM equals 0 and then the second goto WaitHere instruction is skipped. DelayH is then decremented and it will equal 2. The third goto WaitHere will execute because DelayH does not equal 0 yet. This triple loop will continue until DelayH equals 0 which causes the third goto WaitHere instruction to be skipped and then the 500mS delay is complete. This routine does not cause an exact delay of 500mS but it is close enough for our purposes.

The next task is to place this delay routine into the code we have so far.



```

Title    "mICro's Flash Program"

list p=16f84          ; processor type

TurnOnLed    equ 0x00          ; value to turn on LED with RA4
TurnOffLed   equ 0x10          ; value to turn off LED with RA4
PortA        equ 0x05          ; PortA RAM address
TrisA        equ 0x85          ; TRISA RAM address
RA4          equ 0x10          ; PortA RA4 = bit 4
Status       equ 0x03          ; Status RAM address
RP0          equ 0x05          ; Status RP0 bit = bit 5
DelayL       equ 0x20          ; delay register LOW byte
DelayM       equ 0x21          ; delay register MID byte
DelayH       equ 0x22          ; delay register HIGH byte
;
; -----
; PROGRAM START
; -----
;

org 0h          ; startup address = 0000

bsf Status,RP0  ; set RP0 for RAM page 1
bcf TrisA,RA04  ; set RA4 = output
bcf Status,RP0  ; set RP0 for RAM page 0

start          movlw TurnOnLed    ; turn on LED on RA4
               movwf PortA

               clrf DelayL        ; clear DelayL to 0
               clrf DelayM        ; clear DelayM to 0
               movlw 3h           ; set DelayH to 3

```



```

                                movwf DelayH
Wait1    decfsz DelayL           ; subtract 1 from DelayL
                                goto Wait1       ; if not 0, goto Wait1
                                decfsz DelayM     ; subtract 1 from DelayM
                                goto Wait1       ; if not 0, goto Wait1
                                decfsz DelayH     ; subtract 1 from DelayH
                                goto Wait1       ; if not 0, goto Wait1

                                movlw TurnOffLed  ; turn off LED on RA4
                                movwf PortA      ; turn LED off

                                clrf DelayL       ; clear DelayL to 0
                                clrf DelayM       ; clear DelayM to 0
                                movlw 3h         ; set DelayH to 3
                                movwf DelayH
Wait2    decfsz DelayL           ; subtract 1 from DelayL
                                goto Wait2       ; if not 0, goto Wait2
                                decfsz DelayM     ; subtract 1 from DelayM
                                goto Wait2       ; if not 0, goto Wait2
                                decfsz DelayH     ; subtract 1 from DelayH
                                goto Wait2       ; if not 0, goto Wait2

                                goto start        ; do this loop forever

                                end

```

Our code now matches the flow chart we created for this program. A label that defines `TurnOffLed` has been added, and notice how the second 500mS delay has a `Wait2` label. Remember we can't have 2 labels defined with the same name.

Do you notice the value that was used to define `TurnOffLed`?

```
TurnOffLed equ 0x10      ; value to turn off LED with RA4
```

Pin RA4 is pin 4, but `TurnOffLed` is defined as 10h or 16 decimal. How can this be?

Remember RA4 is pin number 4 from Port A, which can also be said as being bit 4. When bit 4 is represented in Binary, it becomes 10000, which is 0x10.

Do you also notice something about the 2 delay routines?

Apart from a different Label name, they are exactly the same code, and they both do exactly the same thing. Wouldn't it be nice to use only one of these delay routines instead of two. It would save us some typing and it would also conserve memory space inside the PIC. These devices have a limited amount of memory, so it is in our best interests to write our code as compact as possible. That way we can do more with the resources we have available.

We can change these two routines into a one by turning them into a single SUBROUTINE. A subroutine is a piece of code that is used many times by different parts of your program. We create these because it becomes wasteful to have the same code copied many times as you need it.

To use a subroutine, we use the instruction `CALL` followed by a label name. This tells the processor to remember where it is now and jump to the part of memory where the subroutine is located. After the subroutine is completed we use the `RETURN` instruction to tell the processor to jump back and continue on from where it was before it jumped to the subroutine. You can learn how the PIC does this by looking at the [MicroPost](#) tutorial software.

This is how we can turn the delay code into a subroutine. Don't forget to add the `/R` in the comment on the first code line, which will be explained later.

```
Delay500    clrf DelayL                ; /R clear DelayL to 0
            clrf DelayM                ; clear DelayM to 0
            movlw 3h                   ; set DelayH to 3
            movwf DelayH
Wait1       decfsz DelayL              ; subtract 1 from DelayL
            goto Wait1                 ; if not 0, goto Wait1
            decfsz DelayM              ; subtract 1 from DelayM
            goto Wait1                 ; if not 0, goto Wait1
            decfsz DelayH              ; subtract 1 from DelayH
            goto Wait1                 ; if not 0, goto Wait1
            return                     ; finished the delay
```

As you can see the subroutine is exactly the same as the original code. The only difference is a Label called `Delay500` which defines the name of the subroutine, and the `RETURN` instruction placed at the end. Now each time we need to use a 500mS delay anywhere in our code all we have to do is use this code line.

```
call Delay500                ; execute a 500mS delay
```

The delay subroutine will save us 9 code lines each new time we need a delay. We can also use the `BSF` and `BCF` instructions to turn the LED on and off which will shrink the code by a further 2 lines. We can also change the `RA4` label to `LED` so it is even easier to read.

```
start       movlw TurnOnLed           ; turn on LED on RA4
            movwf PortA

start       bcf PortA,LED              ; turn on LED on RA4
```

If you can find ways to make your code more compact you may find that it operates much more efficiently and you can fit more code into the chip.

This is the final version of our LED flash code. We have dispensed with the TurnOnLed and TurnOffLed Labels and added BSF and BCF instructions.

```

Title    "mICro's Flash Program"

list p=16f84      ; processor type
;
; The purpose of this program is to make a LED turn on and off
; The LED is connected to PortA pin RA0
; The flash rate is 500mS
;
PortA     equ 0x05      ; PortA RAM address
LED       equ 0x04      ; PortA RA4 = bit 4 for LED
TrisA     equ 0x85      ; TRISA RAM address
TrisB     equ 0x86      ; TRISB RAM address
Status    equ 0x03      ; Status RAM address
RP0       equ 0x05      ; Status RP0 bit = bit 5
DelayL    equ 0x20      ; delay register LOW byte
DelayM    equ 0x21      ; delay register MID byte
DelayH    equ 0x22      ; delay register HIGH byte
;
; -----
; PROGRAM START
; -----
;
org 0h      ; startup address = 0000

bsf Status,RP0 ; set RP0 for RAM page 1
clrf TrisA    ; all PortA = outputs
clrf TrisB    ; all PortB = outputs
bcf Status,RP0 ; set RP0 for RAM page 0

start      bcf PortA,LED ; turn on LED on RA4
           call Delay500 ; execute a 500mS delay

           bsf PortA,LED ; turn off LED on RA4
           call Delay500 ; execute a 500mS delay

           goto start   ; do this loop forever
;
; -----
; SUBROUTINE: waste time for 500mS
; -----
;
Delay500   clrf DelayL ; /R clear DelayL to 0
           clrf DelayM ; clear DelayM to 0
           movlw 3h    ; set DelayH to 3
           movwf DelayH
Wait1      decfsz DelayL ; subtract 1 from DelayL
           goto Wait1   ; if not 0, goto Wait1
           decfsz DelayM ; subtract 1 from DelayM
           goto Wait1   ; if not 0, goto Wait1
           decfsz DelayH ; subtract 1 from DelayH
           goto Wait1   ; if not 0, goto Wait1
           return      ; finished the delay

end

```

You may have noticed that the program made both Port A and Port B pins set as outputs. If the pins are left as inputs with nothing connected to them they will float and may cause damage to the chip.

Floating inputs can also cause the chip to run erratically. By making them outputs, and because we have not written a value to PortA or PortB, the pins will be set at a random logic 0 or logic 1 level when power is applied to the chip. There is nothing connected to any pin other than RA4 with this project so we will do no harm.

This program is available in the software directory and is called `flash.asm`. You can run Mpasmwin again and assemble this code if you like. Then you can use Notepad to see the listing and HEX files produced.

There are many more things to learn when using assemblers, such as Macro's, defining code blocks, constants and others, which are explained in the help file for Mpasmwin or in MPLAB.

It is very hard to remember everything all at once, so just take your time and learn these things as you need them. Have a brief look through the help files so that you have an idea of what to expect and to get a basic knowledge of it's workings, directives and error messages.

MicroPlan Assembler

The MicroPlan beginners assembler can let you write programs just by using the mouse. There is no need to type text other than comments and labels, and you do not need to assemble the code after you finish creating it. The code is assembled as you go.

To start [MicroPlan](#) you can click on 'START - Programs - mICros - MicroPlan'. You may even like to create a short cut.

To make code entry as easy as possible a small prompt window is included under the operand panel on the right side of the screen. This tells you what part of the code line the assembler is expecting you to enter. You will also notice that some text headings are **Red** in colour. These either indicate where you can select the next piece of code data from or which data panel that piece of code will go to when you enter it.

All of the basic Labels that are used with a PIC 16F84 are already defined when you start MicroPlan so you do not need to create these yourself.

However, you do need to create new Labels for new RAM locations and ROM addresses. This is easy to do and will be explained shortly.

The easiest way to see how this program works is to start writing code, so let's begin with the program that we wrote with a text editor.

Press the **New** button to clear the code window.

If you remember, this is the first code line from the LED flash program.

```
bsf Status,RP0           ; set RP0 for RAM page 1
```

The prompt window will have the message **Expecting Label or Command**. This means that the assembler will only accept a Label or an Instruction at this point.

There is no address label on this line of code, so we need to enter the `BSF` instruction first. You can see that each instruction has it's own button on screen, so to add a `BSF` instruction all you need to do is click on the button called **BSF**.

The `BSF` instruction now appears in the Command Panel.

Notice the Code Construction Panel now has the binary value of the `BSF` instruction inserted into the code word, `01 01bb bfff ffff`. The 'b' values represent the bit that will be set to Logic 1 (0 - 7), and the 'f' values represent the RAM register that will be used. The assembler does not know these values yet, because you have not chosen them.

The prompt window will have the message **Expecting RAM Address** and the **RAM List** heading is now a red colour indicating that we can use a label from here. Our instruction is `bsf Status,RP0`, so just click on the **STATUS** label from this list.

The RAM address label `Status` now appears in the Operand Panel, and the `fff ffff` in the Code Construction Panel has been replaced with the binary value of the `Status` address which is 3. It now reads `01 01bb b000 0011`. but the assembler still does not know which bit value to set as yet.

The prompt window will have the message **Expecting Constant** and the **CONSTANTS** heading is now a red colour indicating that we can use a label from this list. Notice that the keypad digits **0 - 7** are also highlighted in red, indicating that we can also choose the bit value from here. We know that `RP0` is bit 5 in the `Status` register, so we can just click on digit **'5'** if we wish.

It would be nicer to use `RP0`, so look down the Constants List and click on the **RP0** label. Notice that it has the value 5 defined next to it.

The prompt window will have the message 'Expecting OK or Comment' and the binary number in the Code Construction panel is now complete. The 'bbb' has been replaced with 101 and the final instruction value is 01 0110 1000 0011.

You can now enter the comment `set RP0 for RAM page 1` into the comment box in the centre left of screen. You do not need to include the semi colon (;).

Now press OK and the code line will be inserted into the main listing.

The assembler is now waiting for you to add some more lines.

This is the next line of code.

```
clrf TrisA          ; set all PortA = outputs
```

To enter it click on **CLRF** and then the `TrisA` Label in the **RAM LIST**. After that enter the comment `set all PortA = outputs` and press **OK**.

Now you can enter this code line.

```
clrf TrisB          ; set all PortB = outputs
```

To enter it click on **CLRF** and then the `TrisB` Label in the **RAM LIST**. After that enter the comment `set all PortB = outputs` and press **OK**. Now you should be able to enter this line.

```
bcf Status,RP0      ; set RP0 for RAM page 0
```

The next code line has an address label followed by an instruction.

```
start      bcf PortA,LED      ; turn on LED on RA4
```

At this stage we have not defined the `LED` Label, so we will do that now.

Scroll down the **CONSTANTS** list until you find the next empty location. Click on the small 'E' button just above this list and then click on the empty constant location. A dialog box will open allowing you to define the name and value of this Label. Type 4 into the Value box and type `LED` into the Label box, then click **OK**. The new label will be entered into the list.

Now you are ready to enter the code line.

To enter the code line we need to use an address target label called `start` first off, so click on the `START` label in the **Jump Table** and it will then appear in the Label Panel. Notice that the **Jump Table** heading has now turned a black colour meaning we can no longer use these labels anywhere else in this line of code.

The prompt window will have the message **Expecting Command**.

To enter it click on **BCF** and then the `PortA` Label in the **RAM LIST**. Then click on the newly created `LED` label in the **CONSTANTS** list and this will be entered into the code word. Add the comment `turn on LED on RA4`, and then press **OK**.

This is the next code line, but it has a label called `Delay500` which has not been defined yet.

```
call Delay500           ; execute a 500mS delay
```

This type of label is for a `GOTO` or `CALL` address destination so it must be entered into the **JUMP TABLE** list. To do that, click on the small 'E' button above this list, click on the next free location in this list, and enter the label name `Delay500` into the Label box that appears, then click **OK**. The new label will appear in the list.

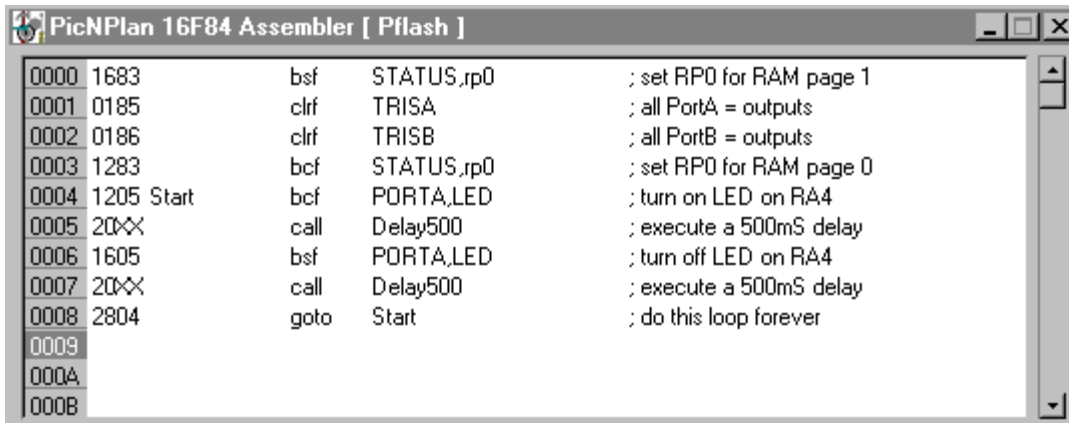
Now click on the **CALL** button followed by the new `Delay500` label, then enter the comment `execute a 500mS delay`, and then click **OK**.

What do you notice about the HEX code that was generated? It should read **20XX**. Notice that the `Delay500` label in the **Jump Table** does not have an address associated with it yet. The assembler cannot put a ROM address value into the code word yet because there are none listed anywhere in the code, so the assembler just leaves it as **XX** for now.

You should now be able to enter these next lines.

```
bcf PortA,LED           ; turn on LED on RA4
call Delay500           ; execute a 500mS delay
goto start              ; do this loop forever
```

After you have finished, the code window should look like this.



If the window does not look like this your code has been entered incorrectly and you will have to start again.

The Labels called DelayL, DelayM, and DelayH have not been defined yet, so click on the small 'E' button above the **RAM LIST** window, then click on the free location at address 20h. Enter DelayL into the edit box that appears and click **OK**. Do the same for the other two registers for RAM addresses 21h and 22h, which will be called DelayM and DelayH.

The next code line is the start of the Delay Subroutine.

```
Delay500  clrf DelayL          ; /R clear DelayL to 0
```

Click on the Delay500 Label in the **JUMP LIST**, then click on **CLRF**, and then the newly created DelayL label. After this, enter the comment /R clear DelayL to 0 then click on **OK**. Notice the /R written in the comment. Please add this as well.

After you do this, you should notice that the **20XX** values have been replaced with **2009** values. That is because the assembler now knows the ROM address of the label called Delay500. This is the value **0009** and it has also appeared next to the Delay500 label in the **JUMP LIST**. Have a look at the ROM address number in the code list where the Delay500 subroutine label appears. It too has a value of **0009**. Each line of code listed in MicroPlan will occupy a ROM address inside the PIC chip when it is finally programmed.

You should now be able to enter these code lines.

```
clrf DelayM          ; clear DelayM to 0
movlw 3h             ; set DelayH to 3
movwf DelayH
```


With this next code line you should know how to create the ROM address label `Wait1`, and then add the instruction.

```
Wait1      decfsz DelayL      ; subtract 1 from DelayL
```

After you click on the label `DelayL` in the **RAM List**, you will notice that the **To F** and **To W** buttons will be highlighted in red.

The prompt window will have the message **Expecting Destination**.

Remember that a destination must be specified for this type of instruction. Click on **To F** because you want the result to go back to the RAM register specified. If you choose **To W**, `DelayL` will not decrement because the result of the instruction will always go to the W register.

Enter the comment, `subtract 1 from DelayL`, and click OK to finish this code line.

You should now be able to enter these next code lines which will complete the program.

```
goto Wait1      ; if not 0, goto Wait1
decfsz DelayM    ; subtract 1 from DelayM
goto Wait1      ; if not 0, goto Wait1
decfsz DelayH    ; subtract 1 from DelayH
goto Wait1      ; if not 0, goto Wait1
return          ; finished the delay
```

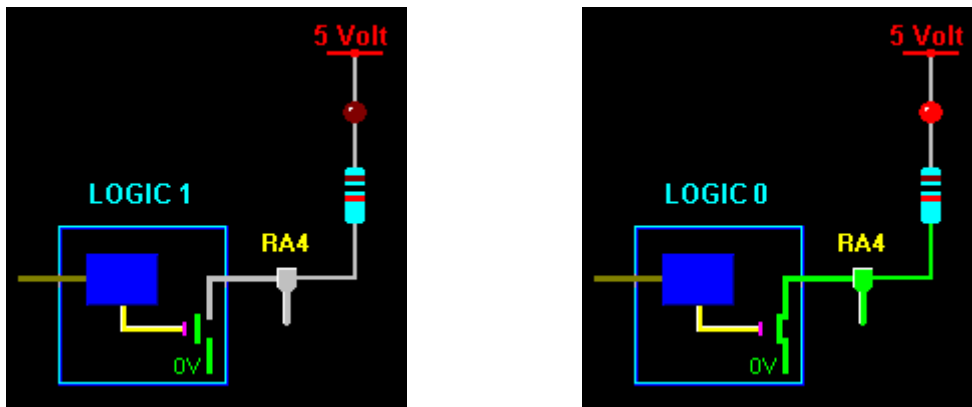
You do not need an `end` assembler directive with MicroPlan, so you can now save this file with any name you like. It is already available in the software directory called `pflash.bls`.

After you save it, and if there are no ROM address labels missing, MicroPlan will generate all the necessary files to program a PIC chip from this code, including a source file. It doesn't produce an error file, because you cannot make a syntax mistake when you enter your code.

Please bear in mind that MicroPlan is a beginners assembler only. It does have all the functionality to generate real PIC code, but if you are going to develop large programs, you will find that a text based assembler is a much better option. Please use MicroPlan to get yourself familiar with PIC code by creating small programs, and then when you become more confident with writing assembler, move into `Mpasmwin` or `MPLAB`. These programs will allow you to get much more productivity out of your time.

Why do you set Port A pin RA4 to Logic 0 to turn the LED on, and Logic 1 to turn it off?

Pin RA4 is an open collector driver when it is set to an output. These two drawings explain how it works.



You can see how the pin can only connect external devices to ground,

More information about port pins can be found in the [MicroPort](#) tutorial.

If you would like to know more about using MicroPlan, see the help file for this program which is activated from a help button on the main screen.

The Simulator

Now that you have your first program completed, should you program a chip and hope that it will work, or should you try to find this out before hand?

It doesn't matter much if you make a mistake with a 16F84, because you can reprogram the chip, but with some PICs, you can't do this. In this case it pays to be sure the code works properly, otherwise you may sacrifice a new chip. These particular chips are One Time Programmable (OTP), and if you program them incorrectly, you will more than likely have to throw them away.

Another programming problem that arises, is that the software appears to work OK for some of the time and then it fails. This type of problem can be very difficult to find, especially when large amounts of code are used for a program.

A way around this problem is to try your program out in a **SIMULATOR**.

These programs allow you to test your code by using a PC to simulate a PIC. Most times they are very helpful and there are a few different types about.

Another benefit of these programs is that they give you another perspective on how your program works.

For example, you can watch RAM registers and how the code affects the values stored in them. You can see how long delay routines take to complete in real time. You can follow the flow of your code and make sure everything works exactly how it is supposed to. You can change the values in registers, and even generate logic levels on the port pins. The best part is, you don't even need to program or use a PIC.

However, there are some drawbacks to using a simulator. Most simulators work at slower speeds than a real PIC processor, so they can take a lot of time to simulate code, especially delays. Sometimes you need to modify your code to get around a delay problem or similar, but this can change the way the software works to the point where it is unusable. Modifying code this way may also fool you into thinking it works, but you may find that in the real world, it doesn't.

The best way to test code in a simulator is break the code down into small routines and try make sure they all work before you put them together to build a complete program.

The LED flash routine that we have created so far is reasonably simple, but what if your code has to control a serial port, as well as drive an LCD display, control stepper motors and read limit switches and a keypad all at the same time? That's quite a task for a PIC and would result in some large and complicated code. The trick is to always break these large problems down into little pieces.

We saw the flowchart earlier which gives us the ability to see our programs pictorially. The blocks that make up the chart may define other flow charts, which in turn may define others as well. A block may need only one code line to create its functionality, but it may also have many. It is up to you the programmer to create the flowcharts, documentation and code so that you or somebody else can totally understand everything that is going on.

To help debug these smaller blocks of code you can use a simulator. After these are known to work, you can begin using these to build up larger programs and you can still use the simulator to test these out. Sometimes a simulator will not show you the whole picture while it is working, especially if you have lots of subroutines and decision making used in the code, so a flowchart is also good here, because most times you can follow it through while monitoring the simulator's progress.

This software package has three types of simulator included with the software. One is called **MicroSim**, another is called **MicroPlay**, and another which has been designed by Microchip is called **MPLAB**.

MicroSim is a beginners simulator and graphically shows the programmer how the code executes inside the PIC 16F84 chip. It accepts code from the list files that are generated when you use a text based assembler like Mpasmwin or from MicroPlan.

MicroPlay works from the list files generated by an assembler as well, but it also allows you to draw a variety of electronic circuits to test your code on. That way you can get an idea of how your program will perform on real life devices.

MPLAB is quite a large PIC development program and with it you can design, develop, test and finally program your PIC chip all within the one user interface. As such, it is out of the scope of this beginners tutorial, so please consult the help files associated with this program when you decide to use it.

Start the [MicroSim](#) simulator and let's see how it works.

The main screen graphically shows the internal structure of the PIC 16F84 chip and you can see most of the special RAM registers which let you control how the chip works.

The box that says `Code space vacant`, is the place that displays your instructions as they are executed. The **PC** box shows the current program counter value and this is the ROM address that corresponds to the currently executing instruction.

If you look around the screen, you will see that all of the values in the boxes are in decimal, and you can change most these by double clicking inside the boxes. This will display a dialog box that shows the current value of the particular RAM address chosen. With this dialog box you can change the value in the RAM location and it's associated label and you can even change how the value is displayed on screen. This may be Binary, Decimal or HEX.

There are three boxes associated with Port A and Port B. The top box in these groups is the data that appears on the pin output latches. The middle box is the data that the PIC reads when it is set as an input, and the lower box is the data for the TRIS register.

Double click on the top PortA box, which is for the [out] data. Now click on the Binary Radio Button in the dialog box that is displayed and click OK.

Now the data that is shown in the [in] and [out] data boxes is displayed in binary. The bit at the extreme right of these values corresponds to Port A pin RA0, or Port B pin RB0.

To view all of the RAM locations, double click on the [address] box in the center of the screen just above the little RAM page book. This will display a complete list of RAM registers. Notice that Port A is displayed in binary. You can also double click on a RAM register value from this list to display the RAM data dialog box again. You will notice that after RAM location 79 and before location 128, each of the values equal zero. That is because the 68 general purpose RAM register addresses which start at address 12, finish at address 79.

You will also see that the TRISA register is at RAM location 133 and should have the value 31 stored in it. TRISB, the next register location, should have the value 255. Double click on each of these registers and change the RADIX to binary as this makes it easier to see the state of any pin. You should notice the main screen values have also changed to binary. At this stage, these pins are set to default input states and as you can see, each input pin is set to Logic 1 in the TRIS register.

Double click on the TRIS A register again and change the binary value from 00011111 to 00011110. This will set Port A pin RA0 as an output.

What do you notice about bit 0 in the Port A [in] and [out] boxes? They should both be the same. Double click on the Port A [out] box and change the Logic state of bit 0. For example, if this bit equals 1, change it to 0 or vice versa. You should notice that bit 0 is still the same in both the [in] and [out] boxes again.

The data acts this way for an output pin because the input circuit is also connected to the pin inside the chip. When a pin is set as an output high, the PIC also 'sees' this logic level when it reads the pin state. The same happens when a pin is set as an output low. You should also notice that the pins set as inputs may have different bit values in the [in] and [out] boxes. When a pin is set as an input, and there is nothing connected to the pin, then there is also nothing to set the pin state. In this condition, the pin is said to be FLOATING.

Down at the bottom left of the screen, you can see how the W register is connected to the rest of the chip through the Arithmetic Logic Unit (ALU).

The ALU is also connected to the rest of the chip via an 8 bit data path and also to the STATUS register by a direct connection. Certain bits change in the STATUS register during program execution. For example if an instruction decrements a value from a RAM register, then the ALU sends a signal to the STATUS register to set or clear the Z bit, or Z FLAG, as it is more commonly termed.

Each time most instructions execute, they can have some effect on the **Carry**, **Digit Carry** and **Zero** bits in this register.

bit	7	6	5	4	3	2	1	0
	IRP	RP1	RP0	TO	PD	Z	DC	C

STATUS REGISTER

Here is a STATUS - Z bit example.

Suppose a register with a label called `Count` has a value of 1 stored in it. If the next instruction is `DECF Count`, then the value will be decremented by 1 and the new value stored here will be 0. Because the result of that instruction was a zero value, the Z flag in the STATUS register is set to Logic 1 by the ALU. If we use the same `DECF Count` instruction again, then the new value stored in `Count` will equal 0xFF. Because the result of this instruction is not zero this time, the Z flag in the STATUS register is now set to Logic 0 by the ALU.

We can see this by using MicroSim's **Tutorial Mode**.

Start this mode by clicking on the `Display - Tutorials` menu item. This brings up a screen showing every instruction that the PIC can execute. Click on **DECF** and a window appears with an explanation for this instruction. After reading the text, click on the arrow button on the lower left of the screen. More explanation text will be shown and finally a screen allowing you to see how the instruction works inside the chip. There should be a small window available that lets you enter a value for RAM location 12. Type in a value of **1** here, and then click on the arrow button again. The screen will show the before and after values of this RAM register for the **DECF** instruction. Keep clicking of the arrow button to progress through the mini tutorial and you will be asked if you want the result placed in the W register or back into RAM location 12. Do you remember the Destination Bit that was mention earlier? This is an example of how it is used. Leave the box **UNCHECKED** for now and click on the arrow button again.

You should now see the data in RAM register 12 taken to the ALU where it is decremented by 1. The result is then placed back into RAM address 12. The decrement result equals zero, so the Z flag in STATUS was set to Logic 1 by the ALU.

Click on the arrow again and repeat the instruction. Don't change the **0** value or the destination bit. This time, the Z flag should be set to Logic 0 by the ALU because the result of the instruction did not equal zero.

Now click on the arrow again and this time, change the Destination check box so that the instruction result goes to the W register. Now keep repeating the instruction. What do you notice about the value in RAM location 12? Did it change at all?

Press **EXIT** to stop the **DECF** mini tutorial, and if you want to see how the other instructions work, please try them out. After you have finished, click on **EXIT** to return to the simulator.

Now we can get back to the LED flash program we wrote earlier and see how it works in the simulator. If you have not assembled it at this stage please do so now or you will not be able to load the code into the simulator.

Click on **File - Open List**. Now choose `flash.lst` from the files listed and click **OK**.

The instruction `bsf Status,RP0` should now be displayed which was the first instruction you wrote in the LED flash source file. Notice it has the value 0000 next to it, and the PC box is displaying 0000. This is showing you that the current program counter value is at ROM address 0000h, and the PIC 16F84 always starts executing instructions from here. Do you remember how we used the `ORG` assembler directive to set the start of the program at ROM address 0h? This was the reason why. If you change this value to something like `ORG 10h`, in your source code and then re-assemble, the assembler will start building the instruction code starting from ROM address 10h. If you try to run the resulting program in a simulator, it will complain because there is now no instruction data at ROM address 0h.

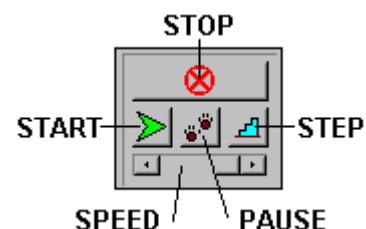
The simulator has some experimental circuits that you can connect to the port pins and these are located in the **Modules** menu.

Click on **Modules - PortA - RA4 OC**.

When you do this, a circuit board will appear connected to the Port A pins. On it, there is a LED and a resistor connected in series from 5 volts to the Open Collector pin RA4. Notice that the LED at this stage is turned off. That is because pin RA4 is still configured as an input. You can change the state of RA4 manually by double clicking on the Port A [TRIS] box and changing bit 4 to 0. Now if the Port A [out] box has bit 4 set as Logic 0, then the LED will light. If this bit equals Logic 1, then the LED will not light. Try changing this bit to Logic 0 to turn on the LED by double clicking on the Port A [out] box.

We are now going to simulate the LED flash program. These are the buttons that control the simulator.

At this stage we will use the **STEP** button which will make the processor execute one line of code at a time. When you first press it, the processor should reset and then start the simulation.



When this happens the LED will turn off because the TRIS registers will be reset so that all the pins are inputs. Make sure the **SPEED** bar is set close to the left side as this is the slowest simulation speed. Now press the **STEP** button.

The `bsf Status,RP0` instruction first causes the processor to get the value stored in STATUS and then place it in the ALU.

Then the ALU sets bit 5 in this value, which if you recall, is the RP0 bit. The result is then placed back into the STATUS register. After this happens, you should see that the RAM page icon near the center of the screen is showing RAM Page 1. Remember that we had to do this first to get access to the TRIS register so that pin RA4 can be set as an output.

Press the **STEP** button again, and the instruction `clrf TrisA` will execute. If bit 4 in the PortA [out] box is Logic 0, the LED will light, otherwise it will stay off.

Press **STEP** again and the next instruction sets all Port B pins to outputs. We have set all the port pins to outputs to stop Floating unused inputs.

Double click on the STATUS box and change the radix to Binary. Press the **STEP** button again and watch the STATUS RP0 bit change back to Logic 0 when the next instruction executes. The RAM Page icon will now change back to RAM Page 0.

For clarity, make sure Port A [out] box is set for binary. Press the **STEP** button again and pin RA4 will be set to Logic 0 by the next instruction. This will cause the LED to come on if it was off previously, or there will be no change if it was already on.

The reason that the LED may come on when the RA4 TRIS pin was set to an output is because the RA4 pin output data was a random value at power on. To gain better control of the port pins, you should set the port pin states to a known value before changing them to outputs. You do this by writing a specific value to PortA and PortB before writing to TRISA and TRISB. In this LED flash program, it does not matter, but it is different if your program communicates with something like an external RAM chip or similar, because it may make a difference to the way these chips operate if they have random data applied to their operating pins.

At this stage take note of the value in the PC box. It should be 0005. Press **STEP** again and the program should then CALL the `delay500` subroutine. Now look at the new value in the PC box. Instead of being 0006, it is 0009. The processor has changed it's program counter and is now pointing to the first code line of the subroutine. The code box will not show this line until the simulator begins executing code again.

The **STACK** box will have the number 0006 in it. This is the ROM address that the PIC needs to start executing from after it has completed the subroutine code. This number is the ROM address following the `CALL Delay500` instruction. If you double click on the **STACK** box you can see the 8 levels of stack that are available in this chip, and at the moment, ROM address 0006 is at the top. When a `RETURN` instruction is executed the most recent address will be **POPPED** from the stack, placed back into the Program Counter and code execution continues from this address. In this case it will be address 0006.

There are animated tutorials on this subject shown in the [MicroPrac](#) program.

Press **STEP** again and watch what happens.

You should see a `Substitute RETURN` instruction listed. The 0006 address value has been **POPPED** from the stack and placed back into the program counter. So why didn't the subroutine code execute?

The answer is in the line of source code that you wrote earlier on. Here it is.

```
Delay500    clrf DelayL                ; /R clear DelayL to 0
```

See the **/R** written at the start of the comment. This is a special Compiler Directive used for the MicroSim and MicroPlay simulators.

Do you remember what was mentioned earlier about simulators being a little bit slower at executing code than a real chip? If we were to use the full 500ms subroutine with the MicroSim simulator you could be waiting for ages to see the LED flash on and off.

By using the **/R** compiler directive, the simulator substitutes the code line it appears on for a `RETURN` instruction. In this way we can bypass the `Delay500` subroutine completely and it appears to us that the code runs much faster.

There are three other Compiler Directives available but they are only recognised from within MicroSim and MicroPlay, and because they are in the comment part of your code, all of the assemblers ignore them. That way the directives don't change the way your source code is assembled for a real chip.

Press **STEP** again, and the next instruction will execute from ROM address 0006. This will turn off the LED because it sets a Logic 1 level on pin RA4.

Now press **STEP** twice to execute the `Delay500` subroutine again. While you do this you will notice the Stack has the address 0008 **PUSHED** into it this time.

This is the ROM address following the second `call Delay500` instruction.

If you remember the source code, the next line should be the one that causes the processor to jump back to the start of the LED flash loop. Press **STEP** and a `goto Start` instruction should be executed. The PC box should have ROM address 0004 in it now, and we are back to the start of the loop.

You can keep pressing the **STEP** button now if you wish, or you can press the **RUN** button to let the simulator run continuously so you can see the results faster. Afterwards, press **STOP** when you have finished.

If you want to see it run a lot faster you can eliminate the animated part of the simulation by clicking on `Display - Visual`. If this menu item is not ticked, the animation is not used. Try it, and press the **RUN** or **STEP** buttons to see the effect.

There are many more options to see with the MicroSim simulator, so look at the help file for this program and don't be afraid to experiment.

If you would like to continue here, please stop the simulation and close the MicroSim simulator program.

Now run the other simulator called [MicroPlay](#).

On the top left corner of the screen there is a button that loads a List File into the program. Click on this and select the `flash.lst` program again. The file name will appear in the top title bar.



Next click on the Load Schematic file button and select the file called `flash.cct`. This will display the LED flash circuit we have been discussing.



Now click on the **RUN** button to start the simulation. MicroPlay will begin executing the code in the `flash.lst` file as well as simulate the electronic circuit on the screen. You will notice that MicroPlay is a lot faster than MicroSim and the LED is now flashing too quickly to see properly.




Stop the simulation by clicking on the **STOP** button.



To slow code execution down, try using the **STEP** button. Each time you press it MicroPlay will execute the code one line at a time.



If you want to see the code lines as they execute, press on the Code List Window button. Each single step line of code that is executed will be highlighted in this window. 

The reason that the simulation seems to run too fast is because the `/R` Compiler directive is still part of the code listing. You should be able to see the `Substitute RETURN` while single stepping and viewing the Code List Window.

To remove this directive, it will be necessary to edit the source code. If you open `flash.asm` using Notepad, find this code line.

```
Delay500    clrf DelayL                ; /R clear DelayL to 0
```

Now delete the compiler directive, save the file again, and use Mpasmwin to reassemble the code.

```
Delay500    clrf DelayL                ; clear DelayL to 0
```

After it has been assembled, you must reload this file back into MicroPlay with the Load List File button and then click on the **RUN** button again. This time it may appear that the LED is not flashing at all, but if you watch the instruction counter at the top right of the screen when it gets to about 600,000 cycles, the LED should change state. From then on, the LED will change state at the same rate.

As you can see these two simulators are a lot slower than real life. There are some simulators around that can operate nearly as fast as a real chip, and if you want one, try doing a web search for PIC simulators. You may even find one for free.

What if you want to watch the RAM register values change in MicroPlay? To do this you need to edit the source file again.

Open it up again with Notepad and find the lines with these RAM label definitions on them.

```
DelayL      equ 0x20                ; delay register LOW byte
DelayM      equ 0x21                ; delay register MID byte
DelayH      equ 0x22                ; delay register HIGH byte
```

Add the `/WD` directives as shown on the following lines, save and reassemble the file again with Mpasmwin.

```

DelayL      equ 0x20      ; /WD delay register LOW byte
DelayM      equ 0x21      ; /WD delay register MID byte
DelayH      equ 0x22      ; /WD delay register HIGH byte

```

This new compiler directive will cause MicroPlay to display these RAM labels in the simulator Watch Window. If you run MicroSim again you will now find these labels listed in the RAM List Window. They will also be displayed in Decimal. If you would like them displayed in Binary, use the `/WB` directive, or for HEX you can use the `/WH` directive.

To view these RAM locations in MicroPlay click on the Watch Window button.



By using MicroPlay you can build many circuits with the components supplied, and it is quite easy to create them.

You can easily change the led flash circuit so that it flashes from pin RA0 instead of RA4. Move the cursor over the wire that connects the LED cathode to RA4. When the mouse is over the wire, the cursor will change to a pair of wire cutters. Now click the left mouse button and the wire will disappear.

Now move the cursor to the yellow connector near pin RA0 and it will change to a cross. Press the left mouse button to start drawing the wire. Each time you need to change direction while running the wire, press the left mouse button and continue. When you have the end of the wire drawn to the center of the yellow connector box on the LED cathode, press the left mouse button to complete the connection. If you press the right mouse button before completing the connection, the wire will disappear.

If you managed to complete this new connection, press the **RUN** button again. You should see that nothing appears to happen. That is because you have not changed the source code to make Port A pin RA0 control the LED.

Open the source file `flash.asm` again and change this code line.

```
LED      equ 0x04      ; PortA RA4 = bit 4 for LED
```

To this.

```
LED      equ 0x00      ; PortA RA0 = bit 0 for LED
```

Now the LED will be controlled by Port A pin RA0.

WHY?

This is one of the code lines that controls the LED.

```
start          bcf PortA,LED          ; turn on LED on RA4
```

The Label `LED` now has a value of 0, not 4. Therefore the BCF instruction changes bit 0 in Port A not bit 4 and this instruction now affects pin RA0 instead of RA4.

You should now go through the comments in the source file and change all the RA4 references to RA0. This is keeping up the practice of good documentation.

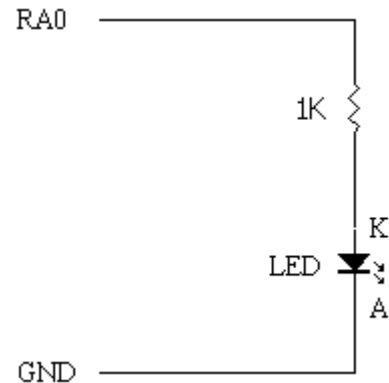
Save the source file and reassemble with Mpsasmwin.

Reload the list file back into MicroPlay and click on the **RUN** button. The LED should now flash as before.


If you want the LED to be on when pin RA0 is logic 1 and off when it is Logic 0 just change the wiring to be the same as the following schematic.

When pin RA0 is at Logic 1, 5 volts is connected to it via switches in the chip and causes 3mA of current to flow through the 1K ohm resistor and LED to ground. The LED is therefore on.

When port pin RA0 is at Logic 0, 0 volts is connected to it via switches in the chip therefore no current flows in the circuit, so the LED is off.




To change the circuit, use the wire cutters to delete all the wiring. Now delete the 5 volt component by placing the mouse over the top of it, hold the CTRL key down and click the left mouse button.

We now need a Ground component and these are located in the component palette at the top of the screen under the heading POWER.  Click on this tab and then click on the ground symbol. A shadow of this component will appear on the work space, so move it around with the mouse until it is in a position near the bottom of the screen directly under the other components. When you are satisfied with the position, click the left mouse button to paste it to the work space.

Now move the LED and resistor down below Port A pin RA0.

You can do this by holding down the SHIFT key and then clicking on the component with the left mouse button. Now connect up the circuit the same as shown in the schematic by drawing the wires as mentioned previously.

Now press the RUN button again, and the LED will begin to flash again, but this time it will be on when the RA0 pin is at Logic 1.


If you want to see the logic level on the port pin, press the STOP button.  Now click on the TOOL tab on the component palette and then on the Logic State tool. Drag the component somewhere near pin RA0 and place it. Now connect a new wire from pin RA0 to the yellow connector on the logic component, and press **RUN**. When RA0 is at Logic 0 the green LED will be lit, and when RA0 is at Logic 1, the red LED will be lit along with the original red LED.


If you want to speed up the LED flashing a bit, try reducing the value that is loaded into the DelayH register. You can also leave out the third code loop within the delay routine altogether.

Just put a **/R** compiler directive on this code line in the Delay500 subroutine.

```
decfsz DelayH      ; /R subtract 1 from DelayH
```

Now the subroutine will end before it decrements DelayH. Try it out and see, but don't forget to reassemble the source code with Mpasmwin.

If you would like to see the digital waveform that is now coming out of pin RA0, stop the simulation and click on the TOOLS tab and then on the Logic Analyzer button. Drag this component into a suitable spot on the screen. 

Now connect a wire from the **A** terminal to pin RA0. Place a 5 volt component on screen near to the Logic Analyser **T** terminal, which is the Trigger input. Now connect the **T** terminal to the 5 volt component. 

Double click on the Logic Analyser and an options window will appear. Change the value in the WINDOW edit box to 10000 uS, and close the options window.

Now press RUN. You should see the square wave that is produced from RA0.

Try placing an inverter between RA0 and the Logic Analyser B terminal.

There are many other components to choose from in MicroPlay. Some are easy to use and some like the LCD display are complicated.

There is a lot of documentation about these components in the MicroPlay help file which is available from the on screen help button. There is also a lot of documentation in the MicroSim simulator help file which you may like to look at. Another good source of information about the PIC 16F84 is the actual data sheet. This is available as a PDF file located on the CD ROM in the ACROBAT directory.

You may have noticed that I am trying to get you to use a text based assembler rather than MicroPlan. You can use MicroPlan if you want to, but it will be of greater benefit the sooner you get used to typing code for assembly. After all, as you may have now started to realise, it is not that hard.

The next stage of the LED flash project is to get it working in the real world. If you have modified the original LED flash program, please restore it to it's original source listing. You can look [back](#) through these notes for it if you like.

Next comes the real test.

The Real World

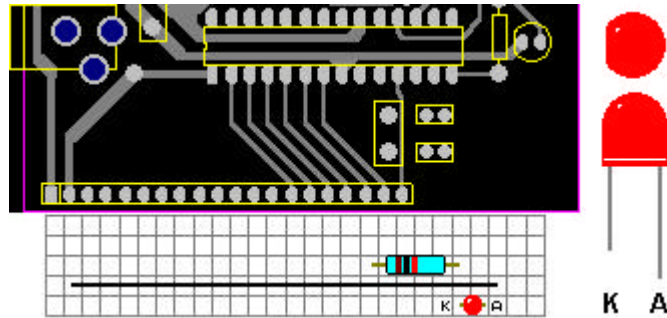
Now comes the stage where you get to test the program in the real world. If the program worked successfully with the simulators, then it should work well in the real world. The only difference will be the time it takes for the delay routine to execute. You saw that it took around 600,000 instruction cycles to complete when using MicroPlay, so in the real world it will take around 600mS with a PIC running with a 4MHz clock speed. I know we were aiming for 500mS, but for this program it does not matter that much.

To make the project come to life we are going to use the MicroPro board to program the chip and provide a means to connect the LED and resistor to the PIC. If you haven't constructed this board, please see [program.pdf](#) for construction details. A 5mm red LED and a 1K ohm resistor should have been supplied with the Experimenters Kit.

The MicroPro PCB is designed to plug into a solderless breadboard which makes experimentation and circuit prototyping easy. Most components can be connected together directly on these boards, but sometimes you will need small insulated wire links. Single strand tin plated telephone wire is excellent for this purpose.

The following diagram shows you how to connect the LED, resistor and MiniPro to the solderless breadboard.

Power should be made available for the MicroPro board, but do not turn it on yet. Connect the MicroPro board to the PC's serial port with a standard serial cable.



When you use any of the **MICRO's** programs that use the serial port, you may see an error message because some other device or program is using the same serial port that the new program is trying to get control of. You cannot have MicroPlay, MicroSim, MicroPro or MicroBasic open at the same time without causing a serial port conflict. Sometimes the mouse causes trouble too.

The default serial port used by the **MICRO's** software programs is COM1. This can be changed by opening a file called `com.dat` which is located in the software installation directory. This is a small text file which tells all of the **MICRO's** programs which COM port to use and the serial baud rate.

The default text is:

```
1
19200
```

You can change the "1" to any port between 1 and 4, but do not change the "19200" baud number. After you have made the COM port change, you can save the file and try running MicroPro again. You may have to do this a few times to get a free com port number.

Once this has been successfully set up, and everything is connected, you are ready to program the PIC chip for the first time.

Chip Differences

The MicroPro board is trying to simulate a PIC 16F84 chip. In order to do this there are a few changes needed in your software.

The general purpose RAM in the 16F84 starts at address 0Ch, but it starts at address 20h in the 16F873. Please use addresses above 20h to 4Fh when using RAM for your program. This corresponds to the upper 48 of the 68 available locations in the 16F84.

You can now apply power to the main MicroPro board. The LED below the PIC processor chip should now be lit. Start the PIC programmer software called MicroPro.

The next task is to select the MicroPro chip for programming.

Click on the **CHIP** detector and scroll down the list that is displayed for an item with this name- `USER`. This selection tells MicroPro that you want to program the PIC chip on the MicroPro board.

Now click on the **LOAD** button and select the file called `flash.hex`. This was the file that you created with Mpasmwin earlier, but make sure this file was created from the original source code listing. If in doubt, check the `flash.asm` file and reassemble it if necessary. After you load the file, you should see that the HEX file contents have been sorted and displayed in a list on screen.

We have now reached the stage where we can program the chip. It seemed an awful long time to reach this point, but don't despair, the procedure gets much faster as you become more experienced.

Now press the PROGRAM button on the MicroPro screen.

The active LED on the MicroPro board and the progress bar should indicate the programming stage. When the programming is complete, a dialog box will be displayed.

Before going any further, do a quick check to make sure the LED and resistor are connected properly on the breadboard.

Now for the big test !!

Press the RUN button which is located in the fuse panel in the MicroPro screen. If all is well the LED should now be flashing on and off every 600mS or so.

If this is the case, jump up and down and scream for awhile and then find someone to pat you on the back.

You have done very very well to get to this point, and you should be proud of your accomplishment.



If the LED is not flashing then don't worry too much. Things like this happen to the best of us, and usually you will find that the solution is quite simple. Sometimes however, the problem can be hard to find so you may have some work to do. Don't let this get you down either, because it's problems like this that makes you a better programmer, and now you will learn much more than someone who got it right the first time.



These are some of the things you could start to look at.

Did the programmer appear to work as expected? If not check it over for problems. Usually MicroPro will generate an error message if it cannot successfully program a device or communicate with the PC.

Check the board for errors such as parts placement and orientation of polarised parts like chips and electrolytic capacitors. Make sure the resistors are in the right positions and check the circuit board for shorts and open circuit connections.

Check the various voltages throughout the circuit.

Make sure the `flash.asm` source code is correct. Re-assemble it and make sure you have chosen the correct HEX file for programming.

Is the LED inserted into the breadboard the correct way around, and are the other connections correct?

If you are positive that everything should be right, try programming again.

If it still doesn't work and you are getting frustrated at this stage, put things aside and have another go later. Sometimes you will spot the obvious after a break. Ask a friend if they can help find the problem. If the project still fails and the PC does not report any errors, you may have a faulty 16F873 chip so you may have to find a friend with a programmer to reprogram the `bpro2.hex` code located in the software installation directory..

I can assure you, the project does work, so please have patience and persist until you find the problem. When you do find it, give yourself a double pat on the back for the extra effort you put in.

If all is well at this stage, you might like to experiment with the software and try different delays and different port pins. After you are happy, you might like to move on to the next project.

Remember, the 16F873 on the MicroPro board is sensitive to static damage, so please treat it carefully.

7 Segment LED display

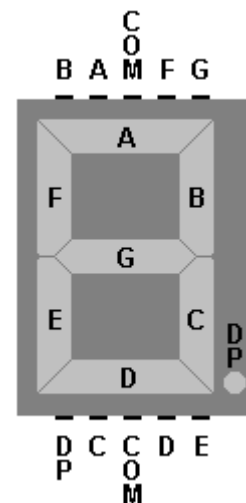
Have you ever wondered how microprocessors can display numbers and other information. Well, wonder no further, because now you are going to find out.

With this project you will learn how to control a 7 segment LED display. These devices are just made out of ordinary LEDs but are arranged in such a way that we can make legible characters out of them when they are energised. All of these types of LEDs can display numbers and some can display the alphabet as well. These are called alphanumeric displays.

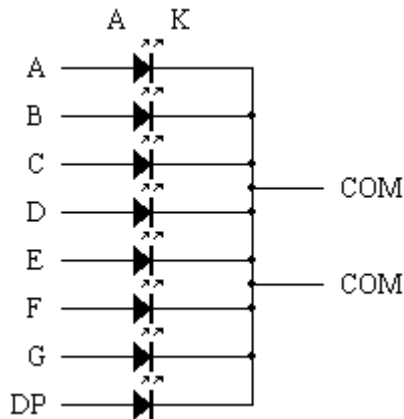
As with any new project, we have to go through the planning stage, so we will start by figuring out the circuit and how it will be connected to the PIC.

There are 7 LED segments used with this type of display, and one extra LED for a decimal point. This makes 8 LEDs in total.

Port A would not be much good to control the display because it only has 5 pins available. Port B on the other hand, is a much better choice because it has 8 pins so we can use one to control each LED. You can also split the LEDs between Port A and Port B if you want to, but that will result in some tricky code. You will soon find out how easy it is to control the display from Port B.



This is the connection diagram for the 7 segment display that is supplied with the Experimenters Kit. It is a Common Cathode type, which means that all of the LED cathodes are connected together within the display and these connections appear at the two **COM** terminals. There are also connector pins shown for each LED segment **A - G** and the **Decimal Point**.

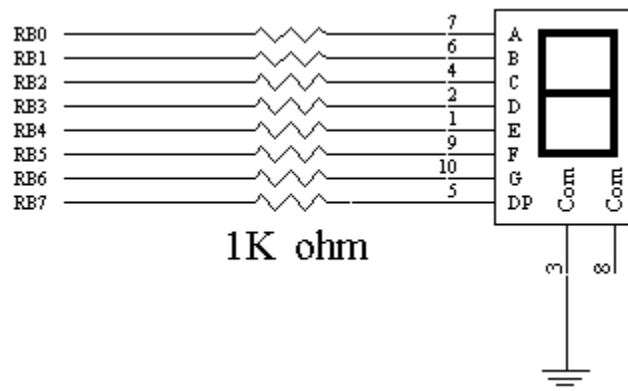


There is another type of display called a Common Anode type, and as you may have already guessed, the LEDs are reversed and all the anodes are connected together.

This is how the common cathode display is connected internally. As you can see there is not much difference with this type of display and a normal LED. There's just more of them.

We can still use the same value 270 ohm series resistor in our new circuit because the LEDs will have the same voltage drop. The only difference is there will be 8 resistors in this circuit, one side of each connected to a LED anode and the other to the Port B pins. There will still be a current of around 10mA flowing in each LED when a port pin is set as an output high, thus turning it on.

This is the circuit we will use to connect the 7 segment display to Port B.



Now that we have the circuit designed, we need to decide what we are going to use the display for. First off, what about a program that will light up each LED segment in turn. We can use a very similar program to the one we used previously.

Here is the start of it.

```
Title    "mICro's 7 Segment Program A"

list p=16f84          ; processor type
;
; This program is used to control a 7 segment display
; The display is connected to PortB pins
; The program energises each segment then repeats
;
PortB      equ 0x06      ; PortB RAM address
TrisA      equ 0x85      ; TRISA RAM address
TrisB      equ 0x86      ; TRISB RAM address
Status     equ 0x03      ; Status RAM address
Carry      equ 0x00      ; Status Carry bit = bit 0
RP0        equ 0x05      ; Status RP0 bit = bit 5
DelayL     equ 0x20      ; delay register LOW byte
DelayM     equ 0x21      ; delay register MID byte
DelayH     equ 0x22      ; delay register HIGH byte
Display     equ 0x23      ; display data
;
; -----
; PROGRAM START
; -----
;

org 0h          ; startup address = 0000

bsf Status,RP0  ; set RP0 for RAM page 1
clrf TrisA      ; all PortA = outputs
clrf TrisB      ; all PortB = outputs
bcf Status,RP0  ; set RP0 for RAM page 0
```

As you can see, it is nearly exactly the same as the start of the previous source file. The only difference is a label called `Display` has been added and some others are missing because we do not need them now. We are not using Port A this time, but we still set all the Port pins to outputs. Remember, we do not want the Port A pins to float as unconnected inputs.

The next task is to energise each LED segment in turn and put a delay in between.

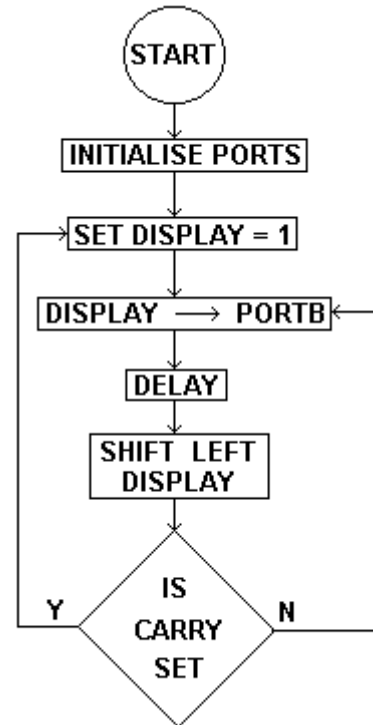
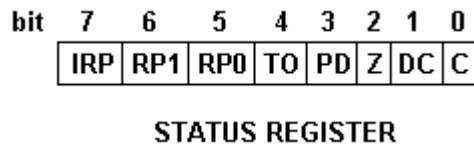
We are going to use the newly defined `Display` register to hold the data that will be written to Port B to turn on each LED.

This is one way we can use the General Purpose RAM locations inside the PIC, which is to store and modify the data that our programs use.

This is a flow chart of the new program.

The value stored in this RAM register is shifted left each time the code loop executes. Eventually the Carry Flag in the STATUS register will be set to Logic 1 and this is the signal to reset the Display value back to one. You can see in the flowchart that there are two loops, and each one gets executed depending on the logic state of the Carry Flag.

There are some new programming concepts we need to know about here. One is the shift left instruction and the other is the Carry Flag.



The **Carry Flag** is bit 0 in the Status register.

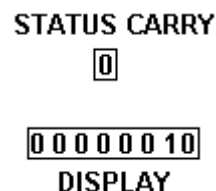
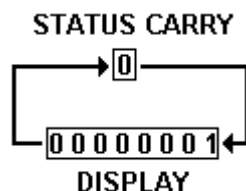
This bit can be automatically set to Logic 1 or cleared to Logic 0 by a number of different instructions. Some of these are ADDWF, SUBWF, BSF, BCF, RRF and RLF. The instruction we are interested in is **RLF** which means to **R**otate **L**eft **F**ile. In PIC terms a **File** refers to a RAM location.

This is how the **RLF** instruction works.

Suppose this is the binary value stored in Display and the Carry Flag is set to Logic 0.

Display = 0 0 0 0 0 0 0 1 Carry = 0

Now a RLF Display instruction executes and each bit in Display is shifted left by one position through the carry bit. The carry bit [0] is shifted into Display Bit 0 and Display Bit 7 [0] is then shifted into the carry bit.



Now suppose this is the binary value stored in `Display`.

`Display` = 1 0 0 0 0 0 0 0 `Carry` = 0

After the `RLF Display` instruction executes, each bit in `Display` is shifted left by one position through the carry bit.



Notice how the carry bit changed to Logic 1 after the instruction `RLF` executed.

According to the flow chart, `Display` is initially set to a value of 1. We can do that by using these instructions.

```
Loop1      movlw 1h           ; set Display value = 1
            movwf Display
```

Now we energise Port B bit 0 to turn on the first LED segment, which is segment **A**. At the moment `Display` has a value of 1 stored in it, which is 00000001 in binary. If we write this value to Port B, that will set pin RB0 to a Logic 1 state which turns on the **A** segment, and Logic 0 to all other bits which turns the other LED segments off.

We can accomplish that by using these instructions. The first one gets the value stored in `Display` and puts it into the W register. The second one sends this value from W to PortB.

```
Loop2      movf Display,w      ; put Display into W register
            movwf PortB        ; now send this value to PortB
```

PortB will now have the binary value 00000001 and LED segment **A** will light.

Now the flow chart says to shift the `Display` value left by one bit position. We use this instruction to do that.

```
rlf Display      ; shift Display value left
```

Remember we could write the instruction with the destination bit included, but because it is missing, the assembler will know that the result is to be placed back into the RAM register, not W.

We can write it like this if we like.

```
rlf Display,f           ; shift Display value left
```

At this stage, the Carry bit will be set to Logic 0 because Bit 7 that came from the `Display` register was zero.

The Carry bit could have been a Logic 0 or a Logic 1 value, because it has not been specifically set previously, and this logic value will have been shifted into `Display` bit 0. Lets assume that it was zero. The carry bit now equals Logic 0 and `Display` equals 00000010.

STATUS CARRY
0
00000010
DISPLAY

We need a time delay next. Do you remember how it was mentioned earlier that we can reuse code blocks from other programs, or from a library of routines we have created? In this program we can use the same `Delay500` subroutine from the previous LED flash program, and to use it we can still use this instruction.

```
call Delay500           ; execute a 500mS delay
```

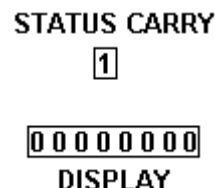
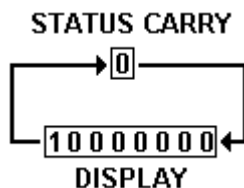
Just remember to redefine the Labels for the registers that the subroutine uses at the start of the program, or the assembler will complain.

According to the flow chart, there are two paths the program can now take depending on the state of the Carry bit. It was set to Logic 0 after the `RLF Display` instruction which means that this bit is not set, so the program branches using the first loop to go back and write `Display` to Port B again.

Now the loop starts again and the new value 00000010 stored in `Display` will be written to Port B. This turns off the LED connected to pin RB0 and lights the LED connected to pin RB1.

As this loop keeps executing, the LEDs will light one by one until the Decimal Point LED is lit. When this happens the `Display` register will have the binary value 10000000 stored in it.

Now when the `RLF` instruction executes, `Display` will have the binary value 00000000 and the Carry bit will be set to Logic 1.



The program will now branch using the second path and loop back to reset the Display value to 00000001. This new value is written to PortB and the process continues forever.

These are the instructions that test the Carry bit and decide on which loop to use.

```
btfss Status,Carry ; test carry bit
goto Loop2         ; carry = 0, do Loop 2
goto Loop1         ; carry = 1, do Loop 1
```

The instruction **BTFSS** means to **Bit Test** a **File** and **Skip** if the bit is **Set**.

`btfss Status,Carry` means to test the Carry bit in the Status register, and if it is set to Logic 1, skip over the following instruction.

If the Carry bit = 0 [clear] when this instruction executes, the following instruction, `goto Loop2` is executed.

If the Carry bit = 1, [set] the `goto Loop2` instruction is skipped over and the `goto Loop1` instruction is executed instead.

You can use the BTFSS instruction on any RAM register including the ports.

```
BTFSS Porta, RA0      ; bit test PortA bit RA0
BTFSS Display,0h      ; bit test Display bit 0
BTFSS Status,z        ; test if any ALU result was 0
```

Here is the complete source listing for this program.

Title "mICro's 7 Segment Program"

```
list p=16f84          ; processor type
;
; This program is used to control a 7 segment display
; The display is connected to PortB pins
; The program energises each segment then repeats
;
PortB      equ 0x06      ; PortB RAM address
TrisA      equ 0x85      ; TRISA RAM address
TrisB      equ 0x86      ; TRISB RAM address
Status     equ 0x03      ; Status RAM address
Carry      equ 0x00      ; Status Carry bit = bit 0
RP0        equ 0x05      ; Status RP0 bit = bit 5
DelayL     equ 0x20      ; delay register LOW byte
DelayM     equ 0x21      ; delay register MID byte
DelayH     equ 0x22      ; delay register HIGH byte
Display    equ 0x23      ; display data
```

```

;
; -----
; PROGRAM START
; -----
;
        org 0h                ; startup address = 0000

        bsf Status,RP0        ; set RP0 for RAM page 1
        clrf TrisA             ; all PortA = outputs
        clrf TrisB             ; all PortB = outputs
        bcf Status,RP0        ; set RP0 for RAM page 0

Loop1    movlw 1h              ; set Display value = 1
        movwf Display
Loop2    movf Display,w        ; put Display into W register
        movwf PortB           ; send this value to PortB
        call Delay500         ; execute a 500mS delay
        rlf Display           ; shift Display value left
        btfss Status,Carry    ; test carry bit
        goto Loop2            ; carry = 0, do Loop 2
        goto Loop1            ; carry = 1, do Loop 1
;
; -----
; SUBROUTINE: waste time for 500mS
; -----
;
Delay500    clrf DelayL        ; /R clear DelayL to 0
        clrf DelayM          ; clear DelayM to 0
        movlw 3h             ; set DelayH to 3
        movwf DelayH
Wait1      decfsz DelayL       ; subtract 1 from DelayL
        goto Wait1           ; if not 0, goto Wait1
        decfsz DelayM        ; subtract 1 from DelayM
        goto Wait1           ; if not 0, goto Wait1
        decfsz DelayH        ; subtract 1 from DelayH
        goto Wait1           ; if not 0, goto Wait1
        return               ; finished the delay

        end

```

Do you think this is has been easy so far? Say yes, you'll feel better.

This is how the 7 segment display should look when the program is executing.



The segments may have a RED colour depending on the part supplied.

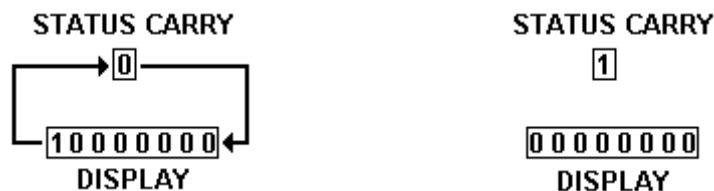
Try this program out in MicroSim. The **/R** compiler directive is still in the delay subroutine so it will be skipped. This file is available in the **MICRO's** directory and is called `seg7a.asm`.

Please assemble this with Mpasmwin and then try it out. When you start [MicroSim](#) and load the file `seg7.lst`, click on Modules - PortB - Seg7, and a 7 segment display will appear for you to use on Port B.

You can also use [MicroPlay](#) if you wish. You can either design you own circuit or there is one in the software directory called `seg7.cct`.

Did the program continue to work properly after the decimal point was lit or did some strange things start happening like multiple LED segments lighting up?

This is what happened after the eighth `RLF Display` was executed. The carry bit was set to Logic 1 this time.



Now when `Loop1` starts again, `Display` is reset to the value `00000001`. What do you think the `Display` value will equal the next time it is shifted left?



As you can see, it now equals `00000011` which is not what we expected or wanted.

Try adding this code line before rotating `Display` left. Reassemble and simulate the file again using `MicroSim`.

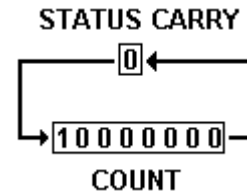
```
bcf Status,Carry    ; clear carry bit to Logic 0
rlf Display          ; shift Display value left
```

Do you have an idea of why the code works properly now? The Carry bit was not always zero when the `rlf Display` instruction was executed. This caused another bit to be set to Logic 1 in the `Display` register which was then written to PortB thus turning on 2 LEDs.

That was a sneaky error, so please remember this.

Always set individual bits or registers to known values before you use them.

As an exercise, try rotating the LED segments the other way around. As a hint, the compliment to the **RLF** instruction is the **RRF** instruction. This rotates the RAM register bits to the right through the Carry bit instead of to the left.



Try using the mini instruction tutorial in the MicroSim simulator to see how the RRF and RLF instructions work. You will also notice during these tutorials, that you have the option of using MicroPrep which shows you how these instructions work on a binary number. Try out [MicroPrep](#) for yourself if you like.

The Counter

You should now be getting an idea as to how you can use code to control the port pins. Just use the TRIS registers to set the pins as outputs, remembering about the RAM page bit, and then just write a value to PortB.

It was mentioned earlier that we can use the LED display to show numbers, so that is what we are going to do now. The good thing is that we can still use most of the code from the previous programs, and a better thing is that we are also going to learn about a concept called a **LOOKUP TABLE**.

A lookup table is a list of data bytes stored sequentially in program ROM, or in RAM. Sequentially just means one after the another and so on.

Why do we use a lookup table?

In this project we are going to make the display count from **0 - 9** and then repeat. In order for the PIC to be able to do that we need ten separate binary values that when written to PortB, will light the correct LEDs to display each digit.

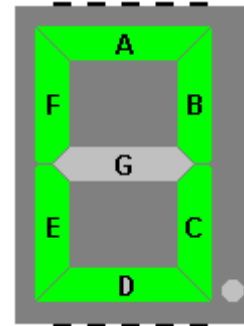
Do you remember that earlier on, it was mentioned that we can use binary values in our source code when we write data to a port? This makes it easier to comprehend because we can instantly see a port bit that will be at 5 volt and those that will be at 0 volt.

Suppose we want to display the digit **0** on the display. How do we define a value to write to Port B that will do that? The answer is surprisingly simple.

This is the LED display showing the digit **0**.

From this diagram you can see that we need to energise segments **A, B, C, D, E** and **F**.

Segment **A** is connected to pin RB0, segment **B** to pin RB1, segment **C** to pin RB2 etc., which means that RB0, RB1, RB2, RB3, RB4 and RB5 must be set to Logic 1 and RB6 and RB7 must be set to Logic 0 for the display to show this digit.



Now that we have figured this out, it is just a simple matter to write the binary value to Port B that sets each of the pins to those logic states.

This is the binary value.

0 0 1 1 1 1 1 1

As we have seen before these are the instructions that allow us to write a value to Port B.

```
movlw 3Fh           ; this is digit 0 value
movwf PortB         ; write it to PortB
```









You can see that it is hard to visualise the digit data when you use a HEX literal value. Lets change it to binary and see the difference.

```
movlw b'00111111'   ; this is digit 0 value
movwf PortB          ; write it to PortB
```

You can instantly see the PortB bits will turn the LEDs on and those that will turn them off. These are the values that will allow us to display all the decimal digits from **0 - 9**.

To create these binary values it is simply a matter of deciding which segments need to be on and then make the corresponding bit value equal 1.

Notice that bit 7 is always 0. That is the decimal point led and we don't need it to come on for this program.

	00111111		01101101
	00000110		01111101
	01011011		00000111
	01001111		01111111
	01100110		01101111

The next task is to figure out how we put that information into a lookup table.

Why can't we just write something like this instead of using a lookup table?

```
movlw b'00111111' ; 0
movwf PortB
movlw b'00000110' ; 1
movwf PortB
movlw b'01011011' ; 2
movwf PortB

etc
```

You can if you want to in this particular program. But what happens if you want to be able to use any of the ten digit values, and not just in a fixed way. With this approach you can't.

When we used a subroutine before, we used a `RETURN` instruction to tell the processor that the subroutine is now completed. There is another instruction that does the same task, but it performs an extra function as well.

It is called the **RETLW** instruction.

This means **RE**Turn from a subroutine with a **L**iteral value in the **W** register.

Suppose we use this instruction to call a subroutine called `DataTable`.

```
call DataTable          ; put digit value in W
```

This will tell the processor to save, or **PUSH**, the next ROM address to the stack and then jump to the code where the subroutine `DataTable` starts.

Now suppose this is the subroutine `DataTable`, and at the moment, the `W` register contents are `0h`.

```
DataTable retlw b'00111111' ; 0
```

Because the next instruction is `RETLW`, the processor knows it is the end of the subroutine and it will pull the top address from the stack, load it into the Program Counter and continue executing code from this ROM location.

When the processor returns from this subroutine, the `W` register contents will have changed from `0h` to `3Fh` which is the HEX value of `0011 1111`.

Now if the next instruction writes the `W` value to `PortB`, the display will show the digit **0**.

```

call DataTable          ; put digit value in W
movwf PortB            ; now display new digit

```

What happens if we add more digits to the `DataTable` subroutine like this?

```

DataTable    retlw b'00111111'    ; 0
              retlw b'00000110'    ; 1

```

Now this may look well and good, but how can we get past the first `retlw` instruction to access the other digit data. If we call `DataTable` the way it is, the subroutine will always return with the digit **0** data in the W register.

The task is to modify the `DataTable` subroutine so that it contains each of the binary values that allows us to display ten decimal digits and also the means to access any one we like.

To figure out how we are going to solve this problem we need to understand how the Program Counter (PC) works. You can see animated tutorials on this subject in [MicroPrac](#), but we will have a brief look at it here.

You should remember that each instruction occupies a single ROM location in the PICs program memory and these start at address 0000h. The PC always has the value of the current instruction, and at power on it is set to 0. After each instruction executes, the PC is incremented by one, unless it was specifically changed by that instruction. As you have already seen, instructions that do this are - GOTO, RETURN and RETLW. The skip type of instructions like BTFSS don't change the value of the PC to skip over the following instruction. If the following instruction is to be skipped, the processor actually gets that instruction but does nothing with it. Then it continues. That is why these type of instructions sometimes take 2 cycles to complete.

Just suppose that the first instruction of the subroutine `DataTable` is located at ROM address 001Ah, (address 26 dec), and the processor executes this next instruction:

```

call DataTable          ; put digit value in W

```

After pushing the current ROM address, plus one, onto the stack, the PC gets loaded with the value 001Ah. The processor will now start fetching and executing instructions from this ROM address.

You may have noticed on the MicroSim simulator screen that a register called **PCL** was attached to the program counter. The **Program Counter Low** register is a RAM register like all the others but it always has the same value as the lower 8

bits of the program counter. It does this because they are both physically connected. From the example above, the PCL register will have the value 1Ah, (26 dec), when the processor is executing the first line of the subroutine.

Start [MicroSim](#) again and load the `flash.lst` file again and run it. You will notice that the PCL stays the same value as the low byte of the program counter which is the box directly below it.

Some examples of this are:

Program Counter	PCL Register
00A0	A0 (160)
0100	00 (0)
03FF	FF (255)
001A	1A (26)

The PCL register is only 8 bits wide, so that is why it only matches the lower 8 bits of the program counter.

As mentioned, the PCL register is a RAM register like all the others and this means you can read and write values to it as well.

If you write a value to this register you also change the lower 8 bits of the program counter, and this makes the processor start fetching from the resulting new ROM address. Some instructions like MOVWF, SUBWF, BSF and ADDWF can be used on the PCL register to change it's value.

If the PC value is 001Ah which is the address of the start of the `DataTable` subroutine code, what would happen if the instruction that is just about to be executed here changed the value in the PCL register? Would this now provide a means to make the processor jump to any `retlw` instruction in the data table? Yes it most certainly would.

This is the complete data table subroutine.

```
DataTable    addwf PCL                ; add W value to PCL
              retlw b'00111111'       ; 0
              retlw b'00000110'       ; 1
              retlw b'01011011'       ; 2
              retlw b'01001111'       ; 3
              retlw b'01100110'       ; 4
              retlw b'01101101'       ; 5
              retlw b'01111101'       ; 6
              retlw b'00000111'       ; 7
              retlw b'01111111'       ; 8
              retlw b'01101111'       ; 9
```


The instruction `addwf PCL` tells the processor to add the contents of W and the PCL registers together and place the result back into PCL. If W equals 0, then nothing gets added to PCL, if W equals 1 then 1 gets added to PCL and so on.

You might think that adding zero to the PCL will cause the processor to keep executing the `addwf PCL` instruction indefinitely. This would be true except for the fact that the program counter is incremented by 1 every time an instruction is executed.

So, if the PCL register value equals 1Ah and the W register value equals 0, then the result of the addition placed into the PCL register is 1Ah.

Then the program counter which still has the value 001Ah is incremented to 001Bh and the `retlw b'00111111'` instruction is executed. This instruction ends the subroutine and loads the W register with the digit **0** data.

If the W register equals 1 when this subroutine is called, the value 1 gets added to PCL which will now equal 1Bh. This also changes the program counter value to 001Bh. Then the program counter is incremented by one so it now has the value 001Ch. This means that the next instruction executed is `retlw b'00000110'` which ends the subroutine and loads the W register with the digit **1** data.

Here is part of the list file that you will create later with Mpsasmwin and shows the ROM addresses we have been discussing. Notice the `RETLW` instruction HEX data. The PIC understands 34XX for this instruction and the XX is the 8 bit value that is made up from your instruction data. If you check digit **0** you will see that binary 00111111 equals 3Fh and that is the first `retlw` instruction.

001A 0782	00072 DataTable	<code>addwf PCL</code>	
001B 343F	00073	<code>retlw b'00111111'</code>	<code>; 0</code>
001C 3406	00074	<code>retlw b'00000110'</code>	<code>; 1</code>
001D 345B	00075	<code>retlw b'01011011'</code>	<code>; 2</code>
001E 344F	00076	<code>retlw b'01001111'</code>	<code>; 3</code>
001F 3466	00077	<code>retlw b'01100110'</code>	<code>; 4</code>

All we need to do to use this subroutine is make sure the W register contains a value between 0 - 9 and after the subroutine executes, the W register will contain the corresponding digit data for the display.

One thing you must make sure of is that the W register does not exceed the value 9. If it does the program counter will be changed to an address higher than the end of the lookup table. This will cause the processor to fetch the wrong instructions and this is definitely not desirable.

We already know how the LED display is going to be connected to the PIC and shown at right is the flow chart for this program. As you can see, it shows what we have just been talking about.

We can use most of the start of the previous program, but we are going to rename the `Display` label to `Pointer` because this register is going to be used as a data pointer for the lookup table. We use the term 'data pointer' because the value in this register is going to be used to point to the information we want to get back from the lookup table. Here is the start of the code.

Title "mICro's 7 Segment Program B"

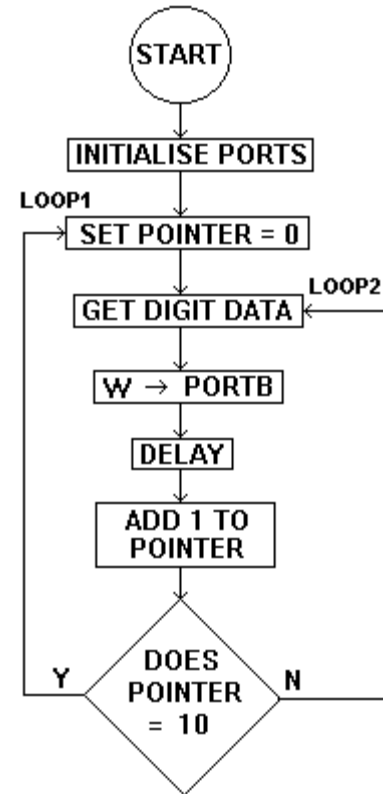
```

list p=16f84          ; processor
type
;
; This program is used to control a 7
segment display
; The display is connected to PortB pins
; The program energises each segment then
repeats
;
PortB      equ 0x06      ; PortB RAM address
TrisA      equ 0x85      ; TRISA RAM address
TrisB      equ 0x86      ; TRISB RAM address
Status     equ 0x03      ; Status RAM address
Z          equ 0x02      ; Status Z bit = bit 2
RP0        equ 0x05      ; Status RP0 bit = bit 5
DelayL     equ 0x20      ; delay register LOW byte
DelayM     equ 0x21      ; delay register MID byte
DelayH     equ 0x22      ; delay register HIGH byte
Pointer     equ 0x23      ; display data pointer
;
; -----
; PROGRAM START
; -----
;

org 0h      ; startup address = 0000

bsf Status,RP0 ; set RP0 for RAM page 1
clrf TrisA    ; all PortA = outputs
clrf TrisB    ; all PortB = outputs
bcf Status,RP0 ; set RP0 for RAM page 0

```



As you may have guessed we can still use the `Delay500` subroutine for the delay. This code is actually going to have two subroutines. One for the delay and another for the lookup table.

After initialising the Ports, we need to set `Pointer` to zero by using this instruction.

```
Loop1      clrf Pointer          ; reset the pointer to 0
```

Now we need to use the lookup table to get the digit data to send to `PortB`. To do this we first need to get the `Pointer` value into the `W` register. Once we have this value set we call the `DataTable` subroutine.

```
Loop2      movf Pointer,w        ; put Pointer into W
           call DataTable        ; put digit value in W
```

Notice how the `Loop1` and `Loop2` labels match up with the two return paths shown in the flowchart.

Now that the 7 segment digit value has been placed into the `W` register by the `DataTable` subroutine we write it to `PortB`.

```
           movwf PortB          ; now display new digit
```

Next we call the `Delay500` subroutine.

```
           call Delay500        ; execute a 500mS delay
```

Now we add 1 to the `Pointer` value. We can do this with a single instruction called **INCF** which means to **IN**Crement a **F**ile register by 1.

```
           incf Pointer         ; add 1 to Pointer value
```

It was mentioned earlier that we must make sure that the `Pointer` value does not exceed 9. If the `Pointer` value equals 10 after being incremented, then we must make sure that it is reset back to zero again, as is shown in the flow chart.

Just how does a processor know when a RAM register is a certain value? The answer is that it doesn't. We have to figure out a way of testing the value ourselves.

Do you remember earlier on, about how the `Z` flag in the Status register will be set to 1 when the result of an instruction equals zero? That is how we can test whether a value is zero or non zero, but it doesn't allow us to test for other values.

To do this task we can use an instruction called **XORWF**, which means to **EX**clusive **OR** the **W** and a **F**ile register together.

This may seem a strange way of testing for a value, but let's have a look at the truth table for the XOR logic function.

A	B	Result
0	0	0
0	1	1
1	0	1
1	1	0

As you can see, if the two bit values A and B are the same, the result of the XOR function equals zero.

So all we have to do is write some code that gets the `Pointer` value and then XOR's it with the value 10. If the `Pointer` value equals 10 then the result will be zero and the Z flag will be set. If the `Pointer` value is not equal to 10 then the result will not be zero and the Z flag will be cleared.

Example:

<code>Pointer = 9</code>	<code>0000 1001</code>	<code>Pointer = 10</code>	<code>0000 1010</code>
<code>XOR with 10</code>	<code>0000 1010</code>	<code>XOR with 10</code>	<code>0000 1010</code>
<code>Result</code>	<code>0000 0011</code>	<code>Result</code>	<code>0000 0000</code>
<code>Z Flag = 0</code>		<code>Z Flag = 1</code>	

This is how we code the comparison function.

First, load the test value (10) into W.

```
movlw d'10'          ; see if Pointer = 10
```

Then XOR it with the `Pointer` value, but we don't want the result to go back to the `Pointer` register because it would corrupt the value stored there.

```
xorwf Pointer,w       ; if it is, reset it to 0
```

Now we test the Z flag in the Status register.

If it is set [1] we know the value in `Pointer` equals 10 so we loop back to the code line that resets this value back to zero. If the Z flag is not set [0] the value is still less than 10 so it is safe to continue with the display code loop.

```
btfss Status,Z        ; test Zero bit
goto Loop2            ; Z = 0, do Loop 2
goto Loop1            ; Z = 1, do Loop 1
```

Now we add the code for the two subroutines and we are finished.

Hands up all those that think this is easy.

Here is the complete code listing for this project.

Title "mICro's 7 Segment Program B"

```
list p=16f84          ; processor type
;
; This program is used to control a 7 segment display
; The display is connected to PortB pins
; The program energises each segment then repeats
;
PortB      equ 0x06      ; PortB RAM address
TrisA      equ 0x85      ; TRISA RAM address
TrisB      equ 0x86      ; TRISB RAM address
Status     equ 0x03      ; Status RAM address
Z          equ 0x02      ; Status Z bit = bit 2
RP0        equ 0x05      ; Status RP0 bit = bit 5
DelayL     equ 0x20      ; delay register LOW byte
DelayM     equ 0x21      ; delay register MID byte
DelayH     equ 0x22      ; delay register HIGH byte
Pointer    equ 0x23      ; display data pointer
;
; -----
; PROGRAM START
; -----
;

org 0h          ; startup address = 0000

bsf Status,RP0  ; set RP0 for RAM page 1
clrf TrisA      ; all PortA = outputs
clrf TrisB      ; all PortB = outputs
bcf Status,RP0  ; set RP0 for RAM page 0

Loop1    clrf Pointer      ; reset the pointer to 0
Loop2    movf Pointer,w    ; put Pointer into W
          call DataTable   ; put digit value in W
          movwf PortB      ; now display new digit
          call Delay500    ; execute a 500mS delay
          incf Pointer     ; add 1 to Pointer value
          movlw d'10'      ; see if Pointer = 10
          xorwf Pointer,w  ; if it is, reset it to 0
          btfss Status,Z   ; test Zero bit
          goto Loop2       ; Z = 0, do Loop 2
          goto Loop1       ; Z = 1, do Loop 1
;
; -----
; SUBROUTINE: waste time for 500mS
; -----
;
Delay500  clrf DelayL      ; /R clear DelayL to 0
          clrf DelayM      ; clear DelayM to 0
          movlw 3h         ; set DelayH to 3
          movwf DelayH
```

```

Wait1    decfsz DelayL          ; subtract 1 from DelayL
         goto Wait1            ; if not 0, goto Wait1
         decfsz DelayM          ; subtract 1 from DelayM
         goto Wait1            ; if not 0, goto Wait1
         decfsz DelayH          ; subtract 1 from DelayH
         goto Wait1            ; if not 0, goto Wait1
         return                 ; finished the delay
;
; -----
; SUBROUTINE: lookup table for display data
; -----
;
DataTable addwf PCL              ; add W value to PC
         retlw b'00111111'      ; 0
         retlw b'00000110'      ; 1
         retlw b'01011011'      ; 2
         retlw b'01001111'      ; 3
         retlw b'01100110'      ; 4
         retlw b'01101101'      ; 5
         retlw b'01111101'      ; 6
         retlw b'00000111'      ; 7
         retlw b'01111111'      ; 8
         retlw b'01101111'      ; 9

         end

```

Sometimes the label names get a bit long and they intrude on the formatting of your program. You can see how this label is getting close to the instruction Mnemonic.

```

DataTable addwf PCL              ; add W value to PC

```

You can rewrite the line like this if you like, and the assembler will accept it as the same thing.

```

DataTable
    addwf PCL              ; add W value to PC

```

I hope you have understood everything that has been mentioned to this point. If you are still a bit confused, try reading through it again or have a look at the [MicroPrac](#) tutorial for a more in depth look. If you want to type this program into a new text file please feel free to do so. It will give you some practice on code writing. Don't forget, you can use [MicroPlan](#) too if you like.

This program is available from the software directory and is called `seg7b.asm`. Assemble the code using `Mpasmwin` and then try it out in `MicroSim` and `MicroPlay` by using the same LED display circuits as before.

As an experiment, try changing this line:

```
movlw d'10'          ; see if Pointer = 10
```

to this:

```
movlw d'11'          ; see if Pointer = 11
```

Reassemble the code and run it with the MicroSim simulator to see what happens.

The program counter will be set to fetch an instruction from the ROM address following the `DataTable` subroutine. The trouble is - there is no code there and the program will crash.

Now try altering the code to make the digits count backwards, both backwards and forwards, or maybe an inverted display. Make the decimal point come on for every odd or even number displayed.

You might also like to try writing code to count this HEX sequence.

0 1 2 3 4 5 6 7 8 9 A b C d E F.

Try connecting the 7 segment display and 8 resistors to PortB on the breadboard and try the `seg7b.lst` program and watch the display count from 0 to 9. Connect the common pins to ground.

Using Switches

It's about time we had a look at using port pins as inputs, so in this experiment we are going to find out how to make the PIC detect to state of a push button switch.

This seems like a simple task, but as we shall see later, writing a program to do the task can get a bit complicated. Up to this point we have been using the port pins when they have been set as outputs. As you may remember from earlier discussions, they can also be set and used as inputs.

To set a port pin as an input, the corresponding TRIS bit must be set to Logic 1. I always remember setting the TRIS bits like this:

1 = Input
0 = Output.

A pin is in a High Impedance state when it is set as an input. This means that it offers a very high resistance to any circuit connected to it, and for most practical purposes, you could say that it becomes an open circuit.

Inside the chip these pins are actually connected to the gates of MOSFETs which is why they have such a high input resistance. If you know anything about these devices, you may also recall that they are very sensitive to Static Electricity, and is the reason you should be careful when handling the chip. Try and avoid touching the pins unless you have some form of static protection.

This is also a reason why you should avoid floating inputs. The stray electrical charges around your body and from other sources can actually affect the sensitive input circuits to such a degree that they cause the chip to malfunction. Problems like these can be very difficult to find if you don't suspect a floating pin.

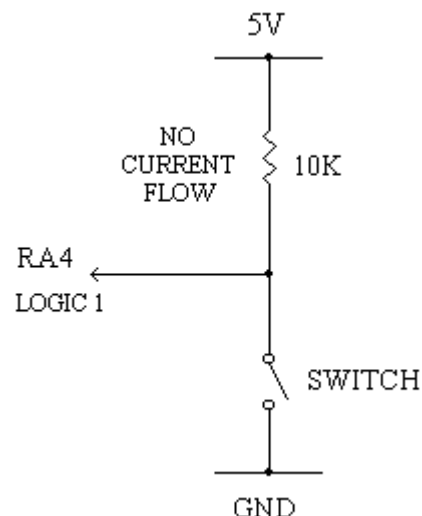
If we look into the input circuit in a simple way, it just detects when a voltage connected to the pin is low enough to qualify for a Logic 0 state, or high enough to qualify for a Logic 1 state.

A voltage somewhere in between may cause the input circuit to oscillate because it can't decide which state to be in. For this reason we should make sure that the circuits connected to the input pins create the correct logic levels. An exception to this is Port A pin RA4. It has a Schmitt Trigger type of input circuit which can cope with voltages that are slowly changing or not strictly at logic levels. These subjects are covered in more detail in [MicroPort](#).

Knowing this information, all we need to do is connect a switch to a port pin, set it as an input, and toggle the switch between 0 volts and 5 volts.

Here is a simple switch circuit and the way it is drawn now, the PIC would detect a Logic 1 on input pin RA4. Remember that the pin is a virtual open circuit, so no current flows through the 10K resistor into the pin. By ohms law, there is no voltage drop across this resistor, so 5 volts must appear on the RA4 pin, which is a Logic 1 level.

A resistor with a value of 10K ohms is fairly common in a switch circuit like this.



When the switch is closed, a 5mA current will flow through the resistor. The GND, or 0 volt rail, will be connected directly to pin RA4, which now becomes a Logic 0 level.

The code that we use to read information from a port is very simple.

```
movf PortA,w    ; read Port A
movf PortB,w    ; read Port B
```

These instructions read all the pin values and place the data into the W register. The RAM page bit [Status,RP0] must be set to Logic 0 for these instructions to work. If this bit is Logic 1, you will read data from the TRISA and TRISB registers instead.

Port A only has five pins so each time you read a value from this register, the upper 3 bits of this value will always be zero.

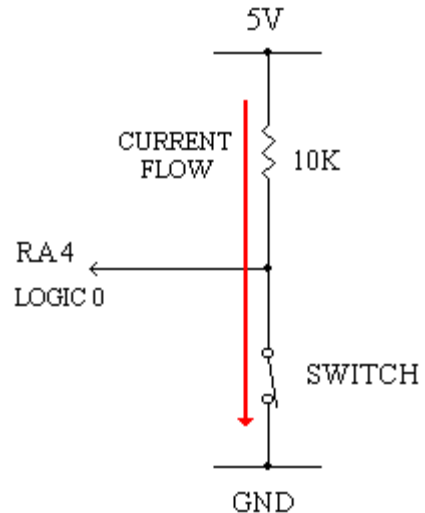
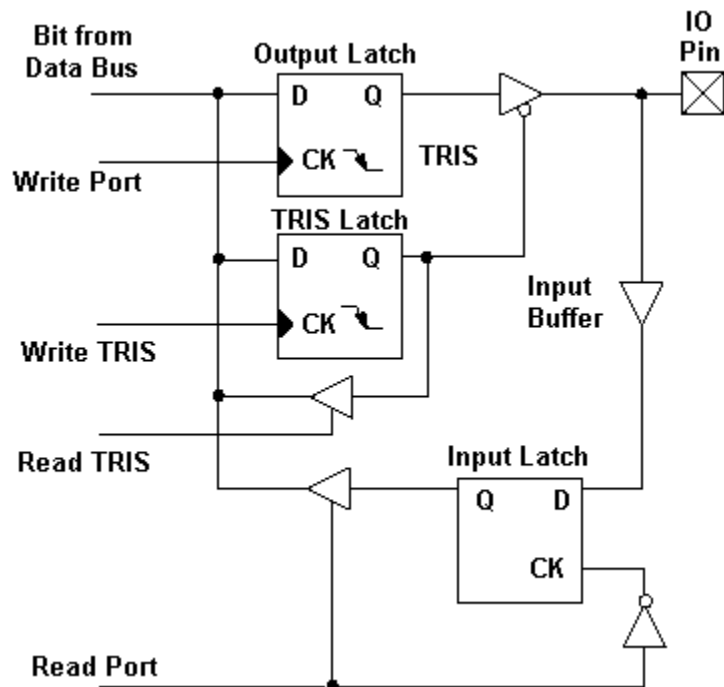
000xxxxx (x = RA0 - RA4 bit value)

You can also write data to Port A or Port B while pins are set as inputs, but only the pins that are set as outputs will change to the new values.

```
movwf PortB     ; write W value to Port B
```

Even though 8 bits of data are written to the Port A register, the upper 3 bits will be lost because there are no pins to write to.

This diagram shows the basic connections for most port pins. As you can see, each port pin has an Input Latch, an Output Latch and a TRIS Latch. These latches are used to store the individual bit information and to provide isolation between the port pins and the internal data bus.



When Port B is written to, the 8 bits of data from the W register are placed into the output latches. When a pin is set as an output, the corresponding latch value is connected to the pin and thus to the outside world. That is why when we write a value to the port it stays there until we change it. The input circuit is always connected to the pin so you can also read the value of a pin set as an output.

When a pin is set as an input, the output latches are disconnected from the pin and therefore have no effect.

The port output latches have random data in them at power up, so we should set them to a known value before we set any pins as outputs. Pins that are left as inputs will adopt the logic values that are set on them by external circuits.

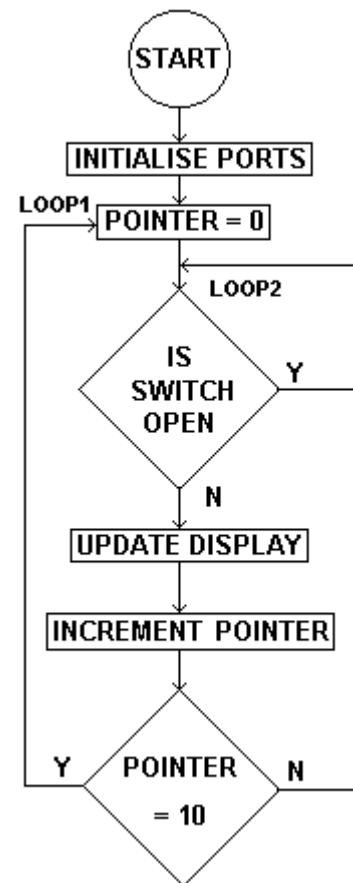
In the previous project we wrote the software so that it automatically incremented the 7 segment display after a time delay. In this project we are going to use a switch to increment the display and we will use the circuit that was shown earlier.

There is one problem that we need to deal with before we start, and that is to understand the nature of mechanical switches. In the world of computers, things operate very fast and a lot of things in the real world seem slow by comparison. When you toggle a switch position, you may think that it is a nice clean transition from on to off and vice versa. Unfortunately, this is not the case. If you drop a tennis ball onto the ground, it will bounce for a while before it comes to rest. The same happens with switch contacts. When they open or close, they actually bounce for a few thousandths of a second before they settle down. This probably doesn't sound like much, but a microprocessor like the PIC can process thousands of instructions in between these tiny bounces.

Because of this, the processor may interpret that a switch is opening and closing many times when in fact you only closed the switch once. The way around this problem is to create some code that ignores these bounces.

To illustrate this we will use this flow chart to construct the new counter program. These flow charts are slowly getting more complex, but it is still quite easy to follow what is going to happen in the program.

There are 3 loops in this program.



According to the schematic, when the switch is open, pin RA4 is at a Logic 1 state. When we read Port A and find that bit 4 equals Logic 1, we know that the switch is open. When bit 4 = Logic 0, we know the switch is closed. These two states will determine the program flow.

If the switch is closed, we update the display and increment the `Pointer` value. If `Pointer` now equals 10, we reset it back to zero again. You should remember this part from the previous program.

The start of the program is exactly the same as before, except this time we need to define PortA and RA4 and we can dispense with the delay RAM registers because at this stage there is no delay routine needed. Instead of defining the port pin as RA4 it has been labelled `Switch`. This will make our source code look a bit clearer.

This time we need to set the RA4 pin as an input to monitor the external switch circuit, but for reasons that we shall see later, we will set all the PortA pins as inputs. The port pins default to all inputs on power on, but it is always safe practice to specifically set them with code.

Port B is used for the display, so all of it's pins are set as outputs. We will also initialise PortB this time, so that when the pins are set as outputs, the display will be blank instead of showing a random value.

```
Title    "Switch Counter Program A"

list p=16f84      ; processor type
;
; This program is used to control a 7 segment display
; The display is connected to PortB
; The program counts from 0 to 9 and then repeats
; each time a switch connected to RA4 is pressed
;
PCL       equ 0x02      ; PCL RAM address
PortA     equ 0x06      ; PortA RAM address
Switch    equ 0x04      ; Switch = PortA pin RA4
PortB     equ 0x06      ; PortB RAM address
TrisA     equ 0x85      ; TRISA RAM address
TrisB     equ 0x86      ; TRISB RAM address
Status    equ 0x03      ; Status RAM address
Z         equ 0x02      ; Status Z flag = bit 2
RP0       equ 0x05      ; Status RP0 bit = bit 5
Pointer    equ 0x20      ; display data pointer
;
; -----
; PROGRAM START
; -----
;

org 0h                      ; startup address = 0000
```

```

        clrf PortB           ; display is blank
        bsf Status,RP0      ; set RP0 for RAM page 1
        movlw b'00011111'   ; set all PortA as inputs
        movwf TrisA         ; the rest are outputs
        clrf TrisB          ; all PortB = outputs
        bcf Status,RP0      ; set RP0 for RAM page 0

```

The start of the Loop1 is the next piece of code and this resets the Pointer value.

```

Loop1      clrf Pointer      ; reset the pointer to 0

```

The next thing to do is test the switch to see if it is open or closed. If the switch is open we must loop back and keep testing it until it is closed. That will be the signal to increment the counter and update the display.

Previously, we used the BTFSS instruction to test the Z and Carry bits in the Status register, but we can use this instruction on the ports as well. However, we need to use the **BTFSC** instruction instead. This means **Bit Test File** and **Skip** the next instruction if the bit is **C**lear. We use this instruction because we want to stay in this code loop until the switch bit [RA4] equals Logic 0.

```

Loop2      btfsc PortA,Switch ; test the switch state
           goto Loop2        ; it is still open

```

As you can see, this code will loop continuously while pin RA4 is Logic 1. When the switch is pressed, RA4 will change to Logic 0, the instruction goto Loop2 is skipped over and the code following is now executed.

This is the code that gets the next display value and then writes it to Port B to update the 7 segment display, and is the same as the previous program.

```

        movf Pointer,w       ; put Pointer into W
        call DataTable       ; put digit value in W
        movwf PortB         ; now display new digit

```

Now we increment the Pointer value and make sure it is always below 10.

```

        incf Pointer         ; add 1 to Pointer value
        movlw d'10'         ; see if Pointer = 10
        xorwf Pointer,w      ; if it is, reset it to 0
        btfss Status,Z      ; test Zero bit
        goto Loop2          ; Z = 0, do Loop 2
        goto Loop1          ; Z = 1, do Loop 1

```

Now add the DataTable subroutine and the program is complete.

```

;
; -----
; SUBROUTINE: lookup table for display data
; -----

;
DataTable addwf PCL          ; add W value to PC
    retlw b'00111111'      ; 0
    retlw b'00000110'      ; 1
    retlw b'01011011'      ; 2
    retlw b'01001111'      ; 3
    retlw b'01100110'      ; 4
    retlw b'01101101'      ; 5
    retlw b'01111101'      ; 6
    retlw b'00000111'      ; 7
    retlw b'01111111'      ; 8
    retlw b'01101111'      ; 9

```

This program is available in the **MICRO's** directory called `counta.asm`, so please assemble it by using `Mpasmwin`.

Now start the [MicroSim](#) simulator and load the program `counta.lst`.

Click on Modules - PortA - Switch and a switch circuit will be connected to Port A.

Now click on Modules - PortB - 7Seg to connect a 7 segment display to Port B.

Each time you use the left mouse button to click on the switch, it will toggle between open and closed. The switch bounce is slowed down quite considerably to allow you to see the effect.

You can also see that the unused Port A pins are connected to the 5 volt rail via resistors. These pins were set as inputs, but they are now all safely tied to a Logic 1 level. If you leave them set as outputs with nothing connected to them, you can save project costs and size because the pullup resistors are not needed.

It is not wise to set the pins as inputs and directly tie them to 5 volts or ground. If the code goes astray for some reason, and the pins accidentally get set as outputs you may have a short circuit problem which may damage the PIC.

You might like to change the Port A [IN] and [OUT] boxes in the simulator to show a binary number instead of decimal. Make sure the switch is in the **OPEN** position and press the **RUN** button.

After the ports are initialized, you will see that the code stays locked in the first code loop because pin RA4 is at a Logic 1 level.

Now press the switch. You will see the contacts bounce, but the code may not detect the switch change until the contacts stay closed. After the software detects the change, it will then update the display and increment the `Pointer`.

Leave the software execute for awhile with the switch closed and the counter will continue to be incremented.

Click on the switch again to open the contacts. After the bounce period, the software will again stay in the switch press detect loop.

Now stop the simulator and click on `Display - Visual`, which will make the code run faster. Now make sure the switch is in the open position, click on **RUN** and operate the switch a few times.

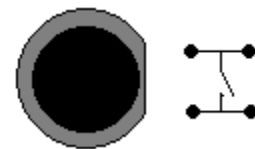
I'll just bet that no matter how hard you try, you will not be able to get the display to increment by one each time you press the switch.

Close MicroSim and run [MicroPlay](#). Then load `counta.lst` and then the circuit file called `switch.cct`. Run the program and you will now find it impossible to increment the counter by 1 each time you press the switch.

Do you understand what is going on here?

The PIC can execute code extremely fast. Even though you have only pressed the switch for a very short time, the PIC has executed the code that increments the display many times. The code as it is now is quite useless in the real world.

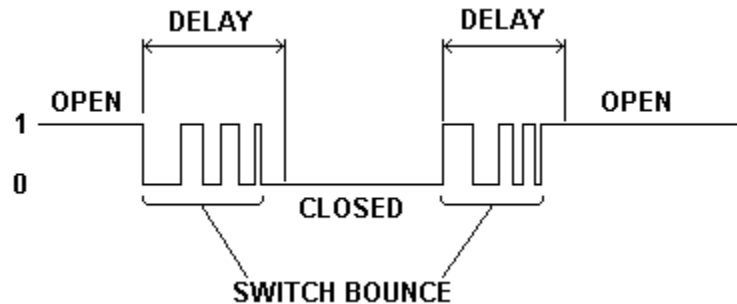
You may now like to connect the switch and display circuits to the breadboard and experiment with them and this code by using one of the simulators and the Real World Interface. You will need some insulated wire to make the connections. This is how the switch supplied with the kit is connected internally and it easily plugs into the breadboard.



You can also program this code into the MiniPro board and connect the circuit to suit. The the code will then run in the real world, and you will still see the problem.

There are two things that need to be done to make the code work as a push button operated counter. One is to create some code that ignores the switch bounce, and the other is to make the code recognise when the switch has been pressed and only increment the counter once for each press.

To solve the debounce problem, we need to make the PIC execute a small delay after the switch has been pressed, or has been released. This delay must be longer than the time that the switch contacts bounce and usually 50mS will be enough. By doing this the PIC should only see the correct logic levels on the switch pin. Here is a diagram explaining the debounce problem.



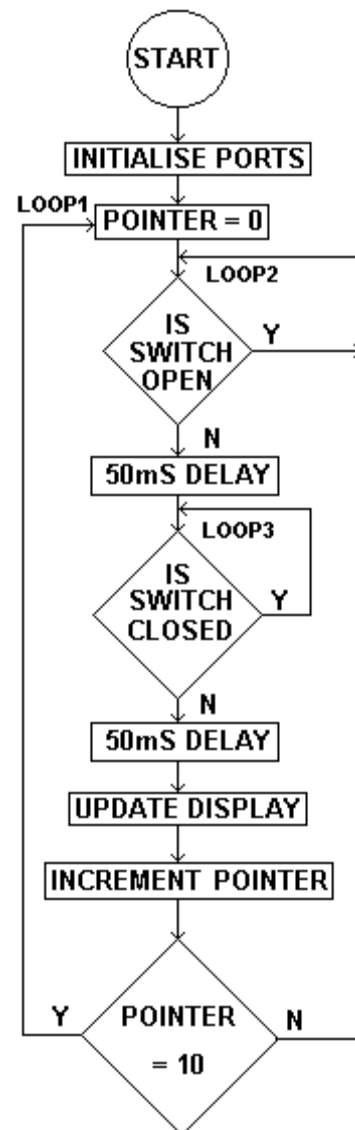
Let's try attacking the problem this way. Suppose we initiate a 50mS delay the instant that the switch press is detected. The switch contacts should now be settled in the closed condition. After the delay, we monitor the switch position waiting for it to release again, and as soon as this condition is first detected, we wait another 50mS. The switch contacts should have then settled in the open position. Now we increment the `Pointer`, update the display and wait for the next switch press and the process repeats.

Here is a flow chart explaining how the software will work. All we have changed from the flow chart shown earlier is the addition of two 50mS delays and another code loop to monitor for a switch release.

The display will not get updated with the next value until the switch has been pressed and then released.

The delay routine is going to be very similar to the one we used previously, but this time it will be much shorter. The software will rely on the fact that the switch contacts should have settled down by the time the 50mS delay is complete. If this is not the case, the software will not work properly and the delay routines will have to be lengthened.

Sometimes you may go through many trial and error periods when you need to determine these and



similar types of operating conditions. An easier way would be to use a storage CRO and view the waveform generated when the switch is opened and closed.

To achieve a 50mS delay using the same time wasting method as shown before, we only need to use two RAM registers, and we can define them like this.

```
DelayA equ 0x21      ; delay counter A byte
DelayB equ 0x22      ; /WD delay counter B byte
```

Now we create a subroutine for this delay by using the following code.

```
Delay50    clrf DelayA          ; /R clear DelayA to 0
           movlw 40h            ; set DelayB to 40h
           movwf DelayB
Wait1      decfsz DelayA         ; subtract 1 from DelayA
           goto Wait1           ; if not 0, goto Wait1
           decfsz DelayB         ; subtract 1 from DelayB
           goto Wait1           ; if not 0, goto Wait1
           return               ; finished the delay
```

This was the code used previously to test if the switch is pressed.

```
Loop2      btfsc PortA,Switch    ; test the switch state
           goto Loop2            ; it is still open
```

This must be followed by the 50mS delay.

```
           call Delay50          ; execute a 50mS delay
```

This is the code to test if the switch is released.

```
Loop3      btfss PortA,Switch    ; test the switch state
           goto Loop3            ; it is still closed
```

This must also be followed by the 50mS delay.

```
           call Delay50          ; execute a 50mS delay
```

Notice the use of the BTFSC and BTFSS instructions to test for opposite switch states.

This is the complete modified switch counter program.

Title "Switch Counter Program B"

```
list p=16f84      ; processor type
;
; This program is used to control a 7 segment display
; The display is connected to PortB
; The program counts from 0 to 9 and then repeats
; each time a switch connected to RA4 is pressed
;
PCL      equ 0x02      ; PCL RAM address
PortA    equ 0x06      ; PortA RAM address
Switch   equ 0x04      ; Switch = PortA pin RA4
PortB    equ 0x06      ; PortB RAM address
TrisA    equ 0x85      ; TRISA RAM address
TrisB    equ 0x86      ; TRISB RAM address
Status   equ 0x03      ; Status RAM address
Z        equ 0x02      ; Status Z flag = bit 2
RP0      equ 0x05      ; Status RP0 bit = bit 5
Pointer  equ 0x20      ; display data pointer
DelayA   equ 0x21      ; delay counter A byte
DelayB   equ 0x22      ; /WD delay counter B byte
;
; -----
; PROGRAM START
; -----
;
org 0h      ; startup address = 0000

clrf PortB      ; display is blank
bsf Status,RP0  ; set RP0 for RAM page 1
movlw b'00011111' ; set all PortA as inputs
movwf TrisA     ; the rest are outputs
clrf TrisB      ; all PortB = outputs
bcf Status,RP0  ; set RP0 for RAM page 0

Loop2    btfsc PortA,Switch ; test the switch state
         goto Loop2         ; it is still open
         call Delay50       ; execute a 50mS delay

Loop3    btfss PortA,Switch ; test the switch state
         goto Loop3         ; it is still closed
         call Delay50       ; execute a 50mS delay

movf Pointer,w  ; put Pointer into W
call DataTable  ; put digit value in W
movwf PortB     ; now display new digit
incf Pointer    ; add 1 to Pointer value
movlw d'10'     ; see if Pointer = 10
xorwf Pointer,w ; if it is, reset it to 0
btfss Status,Z  ; test Zero bit
goto Loop2      ; Z = 0, do Loop 2
goto Loop1      ; Z = 1, do Loop 1
```

```

;
; -----
; SUBROUTINE: waste time for 50mS
; -----
;
Delay50    clrf DelayA                ; /R clear DelayA to 0
           movlw 40h                  ; set DelayB to 40h
           movwf DelayB
Wait1      decfsz DelayA               ; subtract 1 from DelayA
           goto Wait1                 ; if not 0, goto Wait1
           decfsz DelayB               ; subtract 1 from DelayB
           goto Wait1                 ; if not 0, goto Wait1
           return                     ; finished the delay
;
; -----
; SUBROUTINE: lookup table for display data
; -----
;
DataTable  addwf PCL                  ; add W value to PC
           retlw b'00111111'          ; 0
           retlw b'00000110'          ; 1
           retlw b'01011011'          ; 2
           retlw b'01001111'          ; 3
           retlw b'01100110'          ; 4
           retlw b'01101101'          ; 5
           retlw b'01111101'          ; 6
           retlw b'00000111'          ; 7
           retlw b'01111111'          ; 8
           retlw b'01101111'          ; 9

           end

```

This file called `countb.asm` is available from the software directory and should be assembled using `Mpasmwin`.

Start the [MicroSim](#) simulator and load the program `countb.lst`.

Click on Modules - PortA - Switch and a switch circuit will be connected to Port A.

Now click on Modules - PortB - 7Seg to connect a 7 segment display to Port B.

Make sure the Display - Visual, menu item is ticked and press the **RUN** button. The delay subroutine is bypassed so that you can simulate the code in this simulator.

After you have seen what happens with this code, stop the simulator and click on the Display - Visual, menu item and remove the tick. Now when run the

simulator in fast mode you will find that the original switch bounce problem is back. That happens because the delay subroutine has been bypassed.

Close the MicroSim simulator and open the `countb.asm` source file and remove the **/R** Compiler Directive from this code line.

```
Delay50    clrf DelayA                ; /R clear DelayA to 0
```

This is what the new line should look like.

```
Delay50    clrf DelayA                ; clear DelayA to 0
```

Save the file and reassemble it.

Start [MicroPlay](#), load `countb.lst` and then load the circuit file called `switch.cct`. You should notice in the Watch Window that RAM register DelayB is listed in Decimal due to the Compiler Directive in this code line.

```
DelayB     equ 0x22                  ; /WD delay counter B byte
```

This will show you the delay timer as it is counting down. When the simulator is running, you may keep pressing the switch expecting a result to happen when the delay routine is active. The processor is not checking the switch during this time, so you may be fooled into thinking something is wrong.

Press the **RUN** button and see how the program performs.

After you are satisfied, stop the simulation and close MicroPlay.

If you built up the circuit on the breadboard, try using the MiniPro board with this code and verify that it now works in the real world. You will not notice the 50mS debounce delay.

Try modifying the code so that the counter increments only when the button is pressed, not when it is released. Try changing the delay time to see how short it can be before the code starts to fail. If the 50mS delay is not long enough, try lengthening it by initializing `DelayB` with a higher number.

Make the display automatically increment every 500mS when the switch is held down.

If you want to try further experimenting, get an extra button and write some code that increments the counter by pressing one button, and decrements the counter by pressing the other.

My Next PIC Projects

An introduction to the PIC processor II.

miCRO's Next Projects

[Multiplexing 7 Segment Displays](#)

[Relays](#)

[PCLATH](#)

[FSR Tables](#)

[Tables At The Boundary](#)

[Using The EEPROM](#)

[Keypads](#)

Multiplexing 7 Segment Displays

In earlier projects we saw how to drive a 7 segment LED display and make it count from 0 to 9. This simple approach can be quite adequate for some projects, but sometimes there is a need to use more than one display. For example, you may want to display the time, or count the RPM from a car engine, so these projects will require at least four displays.

This is quite an interesting problem. Four displays have 28 segments to drive so how do you drive all of these LEDs when the PIC chip only has 13 Input/Output (IO) pins to use?

The answer is called Multiplexing, and this is where the same segment from each display is driven by only one IO pin. Each of the segments from one display is connected to the same segments from all the others.

You would think that all of the same segments will light up on all of the displays with this approach, but you use a special trick to make sure only one display is active at any one time.

As you may remember, each of the displays has a common ground connection and none of the LEDs will light if this connection is broken. You can make use of this fact to create multiplexed displays.

If only one of displays has a ground connected when the segments are energised, then only this display will show the information needed, and all the others will be off. When you want to display the next digit on the next display, it's simply a matter of turning off the first display, updating the information on the segment pins and then connecting the ground for this digit.

If you do this very slowly, you will see that each digit will be displayed for a short time, but only one at a time. Once the last digit is displayed, you start updating the first again. If you start doing this faster and faster, pretty soon, your eyes can't see the flickering effect, and all the displays seem to on at the same time.

You can drive 4 multiplexed displays from a PIC 16F84 by using 11 IO pins.

- 7 pins for segments
- 4 pins to control the ground connections

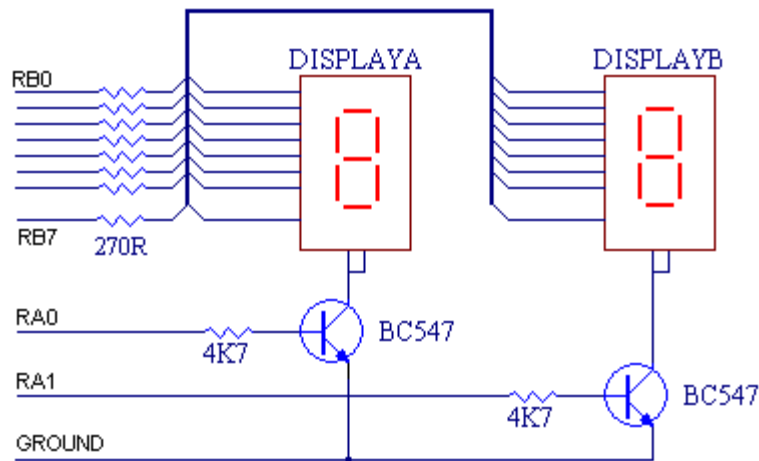
If you want the decimal point included, then you need 12 IO pins.

The PIC can work at very high speeds and it is no problem whatsoever for it to drive LEDs like this.

Here is a circuit for a two digit multiplexed display.

As you can see, each of the segment pins are connected together and also to the PIC IO pins via resistors. These resistors are lower than would normally be used to drive LEDs because the displays are not on all the time and would appear very dim.

The values chosen will allow the LEDs to be seen while multiplexing, but will not be so low that the IO pins are damaged by over current if the processor stopped for some reason and left the LEDs turned on.



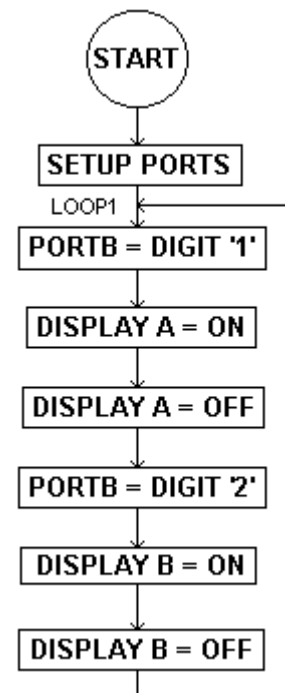
Each of the common connections are connected through a transistor switch to ground. This is a simple way that enables you to switch the displays on and off as the segments are updated. They are controlled by another two IO pins. This method is used because the IO pins by themselves do not have the capability to drive a display module if all of the segments are on at the same time. This would occur while displaying the number 8.

To make this project work, each of the PortB pins are connected to the displays in the same way that was described in the counter example described earlier. The only difference here is that each PortB pin is connected to two display pins instead of one. Pin RA0 is connected to the control transistor 1, and RA1 is connected to the control transistor 2. The decimal points are connected to RB7. Pins RA2 to RA4 are not used in this project, so you should set these as outputs as well to avoid floating inputs.

This is a flow chart that will accomplish a simple multiplexing task for 2 digits.

As you can see, the basic principle is quite easy. First, set the port pins so they can perform the required function. In this case you need to drive the LED segments and the two control transistors, so these pins need to be set as outputs.

The data for showing the digit '1' is put on PortB and then display A is turned on by writing a logic 1 to pin RA0.



This is normally left on for a short time and then display 'A' is turned off again. Now, the data for showing digit '2' is put on PortB and then display 'B' is turned on by writing a logic 1 to pin RA1. After a short time, a logic '0' is written to PortA to turn display B off again, and the process repeats.

Shown below is the code that will accomplish this simple task.

```

Title    "mICro's Multiplexed 7 Segment Program A"

list p=16F84    ; processor type
;
; This program is used to multiplex two 7 segment displays
; The display segments are connected to PortB
; Display A is controlled by PortA RA0
; Display B is controlled by PortA RA1
;
; The program shows the number '1' and '2' on the displays
;
PortA    equ 0x05        ; /WB PortA RAM address
PortB    equ 0x06        ; /WB PortB RAM address
TrisA    equ 0x85        ; TRISA RAM address
TrisB    equ 0x86        ; TRISB RAM address
Status   equ 0x03        ; Status RAM address
RP0      equ 0x05        ; Status RP0 bit = bit 5
DispA    equ 0x00        ; Display A = RA0
DispB    equ 0x01        ; Display B = RA1
;
; -----
; PROGRAM START
; -----
;
org 0h                ; startup address = 0000h

    clrf PortA        ; make sure all displays are off
    clrf PortB        ; make sure all segments are off
    bsf Status,RP0    ; set RP0 for RAM page 1
    clrf TrisA        ; all PortA = outputs
    clrf TrisB        ; all PortB = outputs
    bcf Status,RP0    ; set RP0 for RAM page 0

Loop1    movlw b'00000110' ; segment data for displaying digit '1'
        movwf PortB      ; send it to PortB
        bsf PortA,DispA  ; (0) turn display A on
        nop
        nop
        bcf PortA,DispA  ; (0) turn display A off
        movlw b'01011011' ; segment data for displaying digit '2'
        movwf PortB      ; send it to PortB
        bsf PortA,DispB  ; (1) turn display B on
        nop
        nop
        bcf PortA,DispB  ; (1) turn display B off
        goto Loop1      ; continue and repeat

end

```


You can find this program called `mplexA.asm` in the software installation directory. Please assemble it using `Mpasmwin`.

Start [MicroPlay](#) and load the newly created list file. Now load the circuit file called `mplex.cct`. You will notice that there are no resistors connected to the displays. These particular displays have internal series resistors so you do not need to connect any others, but in the real world you will.

Click on the `Watch` button to display the Watch Window. This should display the values for PortA and PortB in binary. These were included because of the `/WB` directives next to these values in the source code.

Press the `Step` button and after the port setup code has executed you will see how the display data is placed on PortB and then the appropriate display turned on from RA0 and RA1.

Keep pressing the `Step` button until you are satisfied in how the code works.

Now press the `Run` button and you should see how both displays appear to be lit at the same time. You can speed this process up a little by turning off the Watch Window.

MPLAB

I think at this stage it would be good to start using MPLAB from Microchip to develop the software from here on. You should have it installed already so that you could use `Mpasmwin`. Make a short cut to the MPLAB software on the desktop so that you can start it easily.

Start up [MPLAB](#) and click on `File - Open` and select `mplexA.asm` from the software installation directory as the file to load. A window should appear with the source code listed.

Click on `Options - Development mode` and set the processor to **16F84** then make sure the `MPLAB Sim - simulator` item is checked and then press `Reset` to close the window.

Now click on `Project - Build Node` and the software will automatically run MPASM and assemble your code. A dialog box will appear showing the compiler configurations. Just use the default values shown and press `OK`.

Easy isn't it.

Click on the `Step` button on the tool bar at the top of the screen. This is the button with two feet on it. Each time you do this the code will execute one line at a time. Press the `Reset Processor` button to exit this mode.

There are a lot of things you can do in MPLAB and it is quite a large and complicated program. You will not need to worry too much about all the functions it can do. Just learn them as you go and remember to look at the help files supplied.

RELAYS

The PIC has a nominal 25mA current drive capability and this current is quite good for most low power circuits.

Sometimes a component like a motor or light bulb may need to be controlled by a PIC output pin and more than likely, these types of components will draw more than 25mA which can be supplied by the PIC pin.

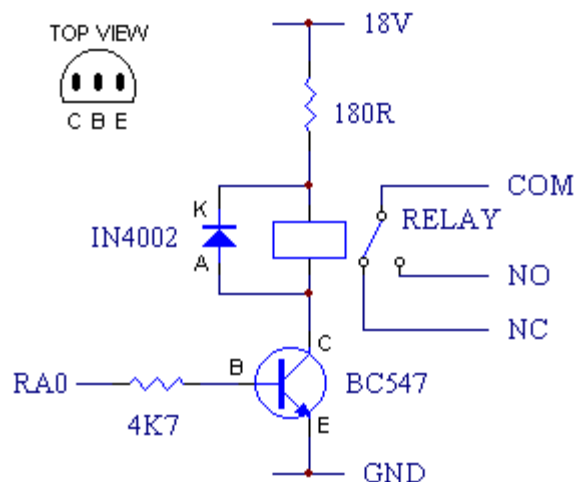
One way around this problem is to use a relay to switch the high currents needed. In some cases, a relay can be connected directly to the PIC, but usually a transistor is used.

A transistor requires only a small base current to let a much larger current flow through it's collector. Also, the component that the transistor is switching can have an operating voltage higher than the 5 volts supplying the PIC. This makes it an ideal component to control relays from the IO pins of the PIC.

Also, by using a relay to provide a switching circuit, any dangerous voltages are isolated from the low voltage digital circuit that the PIC operates in.

Here is the relay circuit.

When the relay releases, the sudden drop in voltage causes a high voltage to appear across the relay coil because of the collapsing magnetic field around it.

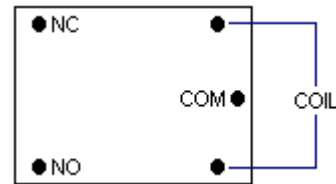


This voltage can destroy the transistor, so some protection is needed for it. This is the reason the diode is connected across the relay coil. When the high voltage is produced, the diode conducts and shunts this voltage back through the relay coil.

The relay supplied with the Experimenters Kit has an operating voltage of 12 volts and a resistance of 400 ohms. The power rail for MicroPro is 18 volts so a resistor is used in series with the relay coil to remove some of the excess voltage.

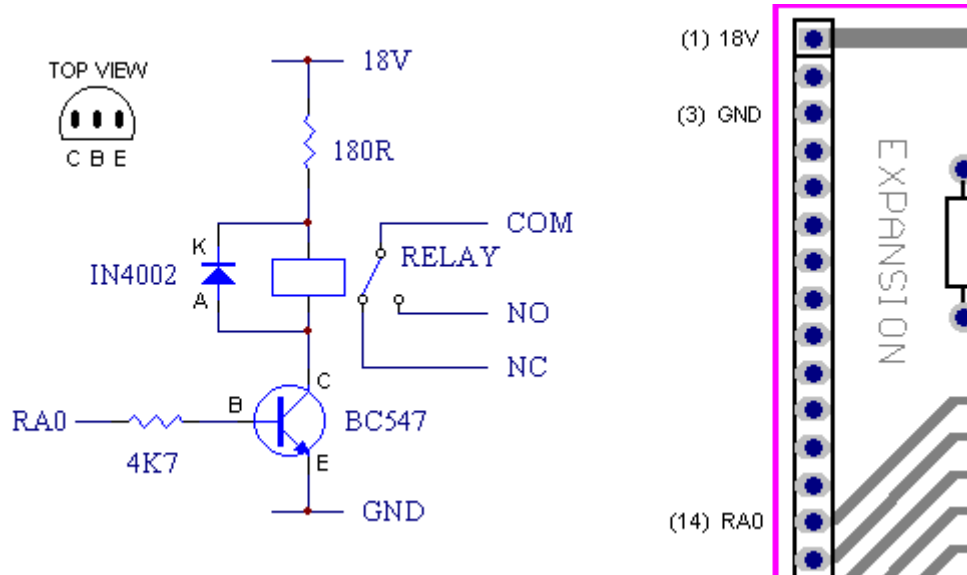
According to Ohms Law, $12 / 400 = 30\text{mA}$ of current should normally flow through the coil. We need to drop 6 volts across the resistor, so $6 / 0.030 = 200$ ohms. The resistor supplied with the kit is 180 ohms which is close enough.

Connect the circuit as shown on a solderless breadboard, or any way that is suitable. Wires may need to be soldered to the relay to provide connections.



Here are the relay connections looking from underneath.

Now connect the circuit to the MicroPro board as shown here.



Start the [MicroSim](#) Simulator and load the file called `relay.lst`. Click on Modules - Project - RealWorld. Run the simulator either by single stepping or in animated mode and the relay should slowly click on and off.

Here is the relay.asm file listing.

Title "mICro's Relay Program"

```
list p=16f84      ; processor type
;
; The purpose of this program is to make a RELAY turn on and off
; The RELAY is connected to PortA pin RA0
; The operate rate is 500mS
;
PortA      equ 0x05      ; PortA RAM address
RELAY      equ 0x00      ; PortA RA0 = bit 0 for RELAY
TrisA      equ 0x85      ; TRISA RAM address
TrisB      equ 0x86      ; TRISB RAM address
Status     equ 0x03      ; Status RAM address
RP0        equ 0x05      ; Status RP0 bit = bit 5
DelayL     equ 0x20      ; delay register LOW byte
DelayM     equ 0x21      ; delay register MID byte
DelayH     equ 0x22      ; delay register HIGH byte
;
; -----
; PROGRAM START
; -----
;
org 0h      ; startup address = 0000

bsf Status,RP0 ; set RP0 for RAM page 1
clrf TrisA    ; all PortA = outputs
clrf TrisB    ; all PortB = outputs
bcf Status,RP0 ; set RP0 for RAM page 0

start       bcf PortA,RELAY ; turn on RELAY on RA0
            call Delay500    ; execute a 500mS delay

            bsf PortA,RELAY  ; turn off RELAY on RA0
            call Delay500    ; execute a 500mS delay

            goto start      ; do this loop forever
;
; -----
; SUBROUTINE: waste time for 500mS
; -----
;
Delay500    clrf DelayL      ; /R clear DelayL to 0
            clrf DelayM      ; clear DelayM to 0
            movlw 3h         ; set DelayH to 3
            movwf DelayH
Wait1       decfsz DelayL    ; subtract 1 from DelayL
            goto Wait1       ; if not 0, goto Wait1
            decfsz DelayM    ; subtract 1 from DelayM
            goto Wait1       ; if not 0, goto Wait1
            decfsz DelayH    ; subtract 1 from DelayH
            goto Wait1       ; if not 0, goto Wait1
            return          ; finished the delay

end
```

Stop the simulator and click on `Display - Visual`. Now click on the `RUN` button to get the simulator to run faster.
Did the relay click on and off?

It probably did, but quite fast.

Start [MicroPlay](#), load the `relay.lst` file and add the RealWorld component to the screen. Make sure the MicroPro board is turned on first. Press the `RUN` button.

Did the relay click on and off?

Probably not, or it did so in an erratic manner.

The reason the relay is doing this is because the PIC is switching it on and off very fast and the relay does not have enough time to react. If you look at the source code you will see the familiar `RETURN` compiler directive `/R` inserted in the delay subroutine.

Open the `relay.asm` file with NotePad or similar and delete the `RETURN` compiler directive and reassemble with MPASM. Now reload it into MicroPlay and the relay should now click on and off every 600,000 counts or so.

Let's see how the program operates in real life.

Close any simulators that may be running and then open [MicroPro](#). Click on the Chip Selector and select `USER` near the end of the list.

Load the file called `relay.hex` and press the `PROGRAM` button. This will write the relay code file into a special area in the PIC memory.

Press the `RUN` button and the relay code will execute and the relay will click on and off about two times per second.

Notice in the source code that the delay counters are defined starting at RAM address 20h.

```
DelayL      equ 0x20          ; delay register LOW byte
DelayM      equ 0x21          ; delay register MID byte
DelayH      equ 0x22          ; delay register HIGH byte
```

In the 16F84, the general purpose RAM registers start at address 0Ch, but in the 16F873 chip on the MicroPro board, these addresses start from 20h. To be able to use the `RUN` feature in MicroPro, you must make sure that the code is compatible with the 16F873 chip.

You may have discovered that you had to turn off the MicroPro board to stop the relay from operating. There is another way you can do this if you are just testing your code.

Load the file called relay2.hex with MicroPro and program it as shown above.

Now press the RUN button.

The relay will click as before, but now you can stop it by pressing the RESET button in MicroPro. You can also restart the code by pressing the RUN button again.

The reason the code acts like this now is because some special code was inserted into the relay code.

When you press the RESET button, the PC sends a byte of data to the MicroPro board. The special code detects this and stops your code from executing and then the MicroPro program takes control. When the RUN button is pressed again, MicroPro starts executing your code again. Here is the special code that is needed to do this task.

```
    btfss PIR1,RCIF          ; check for received PC character
    goto NoChar

    movf RCREG,w             ; Yes, clear receive buffer
    clrf pclath              ; exit run mode
    goto 0h                 ; jump to start of MicroPro code
```

When the RUN button is pressed, the PIC can still communicate with the PC via the serial port. When the RESET button is pressed and the PC sends a byte of data to the PIC which receives and stores it in the RCREG register. When this happens, the RCIF flag in the PIR1 register is automatically set to Logic 1.

The code above tests the RCIF flag and if it is Logic 1, then reads the data in the RCREG and jumps to the code at ROM address 0000h. The code here is the start of the MicroPro software and therefore it takes over and your code stops.

If the RCIF flag is Logic 0, then your code keeps executing at the NoChar address label.

Here is how it looks in the relay2.asm code.

```

                org 0h                ; startup address = 0000

                bsf Status,RP0         ; set RP0 for RAM page 1
                clrf TrisA             ; all PortA = outputs
                clrf TrisB             ; all PortB = outputs
                bcf Status,RP0         ; set RP0 for RAM page 0

start           bcf PortA,RELAY        ; turn on RELAY on RA0
                call Delay500          ; execute a 500mS delay

                bsf PortA,RELAY        ; turn off RELAY on RA0
                call Delay500          ; execute a 500mS delay

                btfss PIR1,RCIF        ; check for received PC character
                goto NoChar

                movf RCREG,w           ; Yes, clear receive buffer
                clrf PCLATH            ; exit run mode
                goto 0h               ; jump to start of MicroPro code
NoChar          goto start            ; do this loop forever
```

If you are very observant you will notice that the relay code starts at ROM address 0000h, so how can the MicroPro code be there as well. The secret is that your code is loaded at ROM address 0800h not 0000h as you may think.

If you were to program a PIC chip in the normal manner with this code it would be programmed starting from address 0000h because that is how it was assembled.

When it is programmed by MicroPro in the special USER mode, the code is actually programmed into the PIC starting from address 0800h. This is a special area set aside in the MicroPro PIC for USER programs.

Did you notice this code line from above?

```
                clrf PCLATH           ; exit run mode
```

If it was not there this line of code will make the PIC jump to address 0800h not 0000h as expected.

```
                goto 0h               ; jump to start of MicroPro code
```

The PIC has ROM pages which occupy 2048 words each and it is the PCLATH that controls which page is accessible to GOTO and CALL instructions. These instructions can only be used within a single ROM page, they can't be used to jump to an address in another ROM page. You must specifically set the PCLATH register if you want to jump to an address that crosses a page boundary.

The PCLATH register is not used for GOTO CALL instructions in the 16F84, but it is in the 16F873.

A ROM page is equivalent to 2K of ROM (2048 words).

16F84 only has ROM space 0000 - 03FFh which is only half a ROM page. (1K)

16F873

Page 0 ROM addresses 0000 - 07FFh (2K)

Page 1 ROM addresses 0800 - 0FFFh (2K)

The next section gives an example on how the PCLATH register works.

PCLATH

PCLATH is an acronym for Program Counter LATch High.

It is used in conjunction with GOTO and CALL instructions and provides a means for the programmer to access any place in the PIC's code memory. The reason you need the PCLATH register to do this comes back to simple binary arithmetic.

Here is the binary representation of the GOTO and CALL instructions.

GOTO

10 1XXX XXXX XXXX

CALL

10 0XXX XXXX XXXX

Each of these instructions is made up of the data that defines the instruction, **101** for GOTO and **100** for CALL.

The other eleven bits **XXX XXXX XXXX** make up the data that defines the target address in the PIC's code memory for these instructions.

You should be aware by now that each instruction occupies a single ROM address in the PIC's memory.

For example:

```
                org 0000h          ; code starts at address 0
Loop            call Delay          ; execute Delay subroutine
                goto Loop          ; repeat forever
                nop
                nop
Delay           nop                ; very small delay subroutine
                nop
                return
```

If you assembled this code, the `Delay` subroutine would start at ROM address 4h, a `NOP` instruction is at address 5h, and the `return` instruction will be at address 6h.

This is how the assembler would define the `call Delay` instruction, because the `Delay` label is at address 0004h.

10 0000 0000 0100

CALL ROM address 4

The HEX code is **2004**.

Start up [MPLAB](#) and click on `File - Open` and select `romadd.asm` from the software directory as the file to load. A window should appear with the source code listed.

Click on `Options - Development mode` and set the processor to **16F873** then make sure the `MPLAB Sim - simulator` item is checked and then press `Reset` to close the window.

Now press `ALT F10` and the software will automatically run MPASM and assemble your code. A dialog box will appear showing the compiler configurations. Just use the default values shown and press `OK`.

Open the file called `romadd.lst` and see that the addresses for the instructions mentioned were as stated. Look at the HEX code for the GOTO and CALL instructions and verify that their target addresses are correct.

Click on the `Step` button on the tool bar at the top of the screen. You will see the code execute line by line and how the CALL and GOTO instructions work.

Close the file(s) and then load and assemble `romadd2.asm`.

You should see a warning message like this appear in the Build Results window.

Message[306] <filename> Crossing page boundary -- ensure page bits are set.

This message is telling you that the target address for a CALL or GOTO instruction has crossed into another ROM page and you should make sure that the PCLATH register has been set correctly.

If the PCLATH register has not been set correctly, then your code will not work as you think.

One other thing to note here, is that your code will assemble with no errors whatsoever, but it will not work in the real world or in a simulator. Messages are not errors, they are there to offer a reminder that something may be wrong with your code.

Here is the listing of `romadd2.asm`.

```
                org 0000h          ; code starts at address 0

Loop            call Delay          ; execute Delay subroutine
                goto Loop          ; repeat forever
                nop
                nop
Error1          goto Error1
;
; Delay subroutine starts at 0800h
;
                org 0800h
;
Error2          goto Error2
                nop
                nop
                nop

Delay           nop                ; very small delay subroutine
                nop
                return
```

Now single step this code and see what happens.

You should see that the code is stuck on this line at address 4h

```
Error1    goto Error1
```

Before we go any further, open both of the list files `romadd.lst` and `romadd2.lst`.

Look at the HEX code that was generated for this code line in both files.

```
Loop      call Delay      ; execute Delay subroutine
```

It should be **2004**. In other words **CALL ROM address 4**.

If you look down the `romadd2.lst` file, you will see the address where `Delay` appears is actually **0804**. So why didn't the subroutine execute?

Have a look at the binary equivalent of **0804**.

0000 1000 000 0100

You need at least 12 bits of information to represent this address.

1000 0000 0100

This amount of bits cannot fit into the `CALL` instruction because there is only enough room for 11 bits of address information available.

In this case the highest bit is ignored by the assembler, so address 0804 becomes 0004, and this information is placed into the `CALL` instruction. That is the reason why both of the `CALL Delay` instructions from each file generate the same HEX code.

1000 0000 0100 (12 address bits)

after assembly, becomes

000 0000 0100 (11 address bits)

1111 0000 0000 0100 (16 address bits)

after assembly, becomes

000 0000 0100 (11 address bits)

This is where the ROM page concept comes from. Eleven bits of information can represent 2048 different addresses.

See how ROM page 1 starts at address 0800.

Page 0	ROM addresses 0000 - 07FFh	(2K)
Page 1	ROM addresses 0800 - 0FFFh	(2K)

OK, so there must be a way of jumping to code in different ROM pages or there is no reason to have the memory available. This means that the PIC must be able to access more address bits from somewhere to make up the bigger address values.

The PIC's we are talking about here have a Program Counter that is 13 bits wide, which means that it can execute code at any addresses ranging from 0 to 8191.

If the GOTO and CALL instructions only have 11 bits available for addresses, where do the other 2 bits come from?

These 2 extra bits come from the PCLATH register. Bits 3 and 4 to be exact.

PCLATH	0000 0000
bits	7654 3210

Every time a GOTO or CALL instruction is executed these two bits are added to the 11 address bits from the instructions to make up the 13 bit number that is placed into the Program Counter.

Example: assuming the Delay label is at ROM address 0804:

PCLATH = 0000 0000

call Delay

PCLATH bits 4 and 3 = 00.

Delay address from instruction = 000 0000 0100.

Total address = 0 0000 0000 0100 (13 bits)

Therefore the Program Counter = 0004 and the code at this address executes, but it is not where we intended.

PCLATH = 0000 1000

```
call Delay
```

PCLATH bits 4 and 3 = 01.

Delay address from instruction = 000 0000 0100.

Total address = 0 1000 0000 0100 (13 bits)

Therefore the Program Counter = 0804 and the code at this address executes correctly.

This is how the PCLATH bits 4-3 should be set to jump to the different ROM pages.

00	Page 0	0000 - 07FFh	2K
01	Page 1	0800 - 0FFFh	2K
10	Page 2	1000 - 17FFh	2K
11	Page 3	1800 - 1FFFh	2K

If you look at the largest PIC memory in this series of chips, you will see that it is 8K which spans 4 ROM pages. Can you guess why?

So, what changes are needed to make the code work?

All you have to do is set the PCLATH so that bits 3 and 4 point to the correct ROM page where the target GOTO or CALL address is located.

To help with this, there is an assembler directive which gets the upper 8 bit value of a ROM address.

Example:

```
movlw High(Delay)
```

In the program called `romadd2.asm`, the `Delay` subroutine starts at 0804h.

This instruction tells the assembler to look up the symbol table for the `Delay` label and get its value. In this case `0804h`. It then gets the upper byte of this value which is `08h` and places it in the `W` register.

When we then write this value to the `PCLATH` register like this, it will be set at the correct ROM page.

```
movwf PCLATH
```

PCLATH = 0000 1000

If you used these instructions with the `romadd.asm` code, the data loaded into the `PCLATH` will be zero.

The `Delay` label in this case is at address `0004h`, therefore the upper byte = `00h`.

Here is the modified code. Type the extra lines into the `romadd2.asm` file, re-assemble, and single step it to verify that it now works.

```
Loop      movlw High(Delay)    ; upper byte address of Delay
          movwf PCLATH
          call Delay           ; execute Delay subroutine
          goto Loop           ; repeat forever
          nop
          nop
Error1     goto Error1
;
; Delay subroutine starts at 0800h
;
          org 0800h
Error2     goto Error2
          nop
          nop
          nop

Delay      nop                 ; very small delay subroutine
          nop
          return
```

Did it actually work properly this time?

If all went according to my devious little plan, the code should have ended up locked in a loop at address `0800h`.

```
Error2     goto Error2
```

Do you know why it didn't go back to the start of the code when the code line `goto Loop` executed?

The `Loop` label is at address 0000h and the `Error2` label is at address 0800h. GOTO and CALL instructions both use the PCLATH register <4-3> bit contents.

Which ROM page was the PCLATH set to after the subroutine returned and the `goto Loop` code line executed?

It was still pointing to ROM page 1 wasn't it.

`goto Loop` in binary is **10 1000 000 000**

The PCLATH value was 0000 1000, so this is the complete ROM address that the PIC computed for the `goto Loop` instruction.

0 1000 0000 0000

This means `goto` address 0800h, not 0000h where we wanted.

For correct program flow, you needed to reset the PCLATH to the ROM page where the GOTO address is intended, in this case ROM page 0. You can do that the same way as before

```
movlw High(Loop)
movwf PCLATH
```

Or, because you definitely know it is ROM page 0, you can simplify the code like this.

```
clrf PCLATH
```

If you don't know which ROM page the target address is located, use the first method to automatically get the value, or you can also look at the *.lst files to see the address where the label is and specifically set the PCLATH with the upper byte value.

Here is the modified listing. Add the extra line, re-assemble and step through the code again. This time it should work properly.

```
Loop      movlw High(Delay)    ; upper byte address of Delay
          movwf PCLATH
          call Delay           ; execute Delay subroutine
          clrf PCLATH          ; ROM page 0
          goto Loop           ; repeat forever
          nop
          nop
Error1     goto Error1
```

When single stepping, view the contents of the W register when the `movlw High(Delay)` instruction executes to verify that it does equal the upper byte value of the `Delay` label address.

Click on Window - Special Function Registers.

During single step, registers are highlighted in **red** when their values change.

If you ever write large programs, be sure to take notice of the assembler warning messages. They are there to help you.

If your code occupies more than a single ROM page, then you must be aware of the value of the PCLATH register when using `CALL` or `GOTO`.

One last thing.

Did you notice that the PCLATH register did not need to be changed for the `RETURN` instruction even though the code jumped back to ROM page 1 from ROM page 0?

The `RETURN` instruction pulls a complete 13 bit value from the STACK and places it in the Program Counter, therefore the PCLATH value is not used.

To view more on ROM addressing, please look at the **ROM Page** and **GOTO - CALL** sections in [MicroPost](#) and the MicroPro help file under **USER PROGRAMS**.

There is also some information in the supplied PDF files for the 16F84 and 16F873/4/6/7 chips.

FSR TABLES

Usually, instructions are executed in the Direct Addressing mode which simply means that the instructions are acting directly on a RAM address.

Example.

```
clrf PCLATH
movwf PORTA
addwf 0x20
xorwf MathReg
```

FSR stands for File Select Register and it's purpose is to allow the programmer to use instructions in the Indirect Addressing mode.

In this mode of operation, the instructions don't communicate directly with RAM addresses. Instead the RAM address data is fetched from the information stored in the FSR register.

Now at this stage you might be thinking, "That sounds great, but what good is it?"

Well, suppose your application needs to specifically initialise a section of RAM at power up so that all values are 0x00. It may be that if you didn't do this your application may crash at some stage or give erroneous results. For instance, this may be the first piece of information that a series of displays show when the project is first turned on.

Let's say you need to clear 4 RAM locations. That's quite easy, and you can do it like this.

```
clrf DisplayA
clrf DisplayB
clrf DisplayC
clrf DisplayD
```

What happens if you need to clear 20, 30 or even 40 RAM registers?

It would become quite wasteful of the memory space if you had to write a `clrf` instruction for each of these registers.

This is a good example of how the FSR register and Indirect Addressing mode comes in handy.

The FSR and Indirect Addressing work in conjunction with another register called the **INDIRECT FILE** register or as it is more commonly called, **INDF**. This register is located at RAM address 0h, and you cannot physically read or write data to this register.

Well, what is the good is that?

Here is where the INDIRECT part of the addressing takes place.

When you write data to the INDF register, the processor gets the value stored in the FSR register and uses that as the RAM address to store the new information in.

When you read data from the INDF register, the processor gets the value stored in the FSR register and uses that as the RAM address to get the new information from. In actual fact the data stored in the FSR register is called a POINTER.

It's like saying...

Hey, I need to deliver a package to someone in a multi-storey building but I don't know where there are, so I'll go and ask the doorman for the information.

The doorman in the PIC's case is the FSR.

Your code can be made to run a lot more efficiently by using the FSR to do some processing on a block of RAM locations, and it also conserves the limited code space which is a must in the microcontroller world.

Here is an example of clearing a block of 10 RAM registers by using the FSR.

Suppose the first RAM location of the block starts at address 32dec, (20h). That means the last address in the block is 41dec, (29h).

Maybe this needs some clarification.

$32 + 10 = 42$, so why does that last address in the block only equal 41?

You have to remember that address 32 is used as the first location in the block, which means there are another 9 locations used, so the last location is at address 41. This can be quite confusing at times. Count it out on your fingers to verify it.

The first thing to do with our routine is to set the FSR to point to the first RAM address of the block.

```
movlw d'32'           ; set start address
movwf FSR
```

Now this line clears the RAM register whose address is stored in the FSR.

```
clrf INDF             ; clear the RAM location
```

From here on we could just simply increment the FSR and clear the next RAM location, increment the FSR, clear, increment etc. This would do the task we want, but it is even less efficient than the first method we tried.

The most efficient way in this case, is to set up a loop. After we have cleared the last RAM location in the block, the loop terminates. This will occur when the FSR value is one greater than the last RAM location. Look at the routine to see how this works.

```
clear    clrf INDF           ; clear the RAM location
         incf FSR            ; set ready for next location
         movlw d'42'         ; test if finished
         xorwf FSR,W
         btfss STATUS,Z
         goto clear          ; not yet, keep going
```

Why didn't we use `movlw d'41'` to test for the last RAM location?

Because then the last location in the block would not have been cleared.

When the last RAM address, 41, is cleared, the FSR is incremented and then equals 42, which ends the loop.

Make sure you understand what is happening in that routine before proceeding.

It seems a fiddly thing to do when we humans have to do the mental arithmetic to find the value for the end of the RAM block, so wouldn't it be nice if we didn't have to worry about doing this.

Well, there is a way. The assembler lets you do some basic arithmetic within your code, and in this way it does the math for you.

Here is another way of doing the same task with 10 address locations in the block.

```
clear      clrfs INDF          ; clear the RAM location
           incf FSR           ; set ready for next location
           movlw d'32' + d'10' ; test if finished
           xorwf FSR,W
           btfss STATUS,Z
           goto clear          ; not yet, keep going
```

Hang on a minute, we can make this even easier.

Remember, that you have to define where in the RAM memory that this block starts from. In this case we said it starts at address 32dec, (20h).

You can define the block of RAM at the start of your code with a CBLOCK statement and use labels instead of actual numbers. This takes the pressure off the programmer who now, doesn't need to worry about the math.

```
                CBLOCK d'32'

RAMBlock: d'10'

                ENDC
```

This tells the compiler to reserve 10 bytes of RAM starting at address d'32'. It can also be written in HEX if you like.

```
                CBLOCK 0x20

RAMBlock: 0Ah

                ENDC
```

You can also have other variables entered into the block just by adding labels. These can appear anywhere in the CBLOCK statement and you can enter any amount as long as there is sufficient RAM available in the chip you are using. The 10 bytes will always be set aside for our block of RAM and we don't need to know what addresses they are located at.

In the past, this is how we have declared RAM variables.

```
Data1      equ 20h      ; data for display 1
Data2      equ 21h      ; data for display 2
Data3      equ 22h      ; data for display 3
Data4      equ 23h      ; data for display 4
RAMBlock   equ 24h      ; 10 byte block of data
Flash      equ 2Fh      ; flash counter
```

You can see that we have had to mentally work out what address `Flash` needs to occupy so that it doesn't get muddled up in the `RAMBlock` of 10 bytes that we need.

With the `CBLOCK` method, it becomes much easier because we let the assembler worry about assigning RAM addresses and making it all fit properly. In this case, `Data1` occupies RAM address `0x20`, `Data2` occupies `0x21`, etc., which is the same as above.

```
                CBLOCK 0x20

Data1           ; data for display 1
Data2           ; data for display 2
Data3           ; data for display 3
Data4           ; data for display 4
RAMBlock:0Ah    ; 10 byte block of data
Flash           ; flash counter

                ENDC
```

Now you can rewrite the clearing code without worrying about the RAM addresses that the registers occupy. However, you still need to know there are 10, (0Ah) bytes to clear in the block.

```
                movlw RAMBlock           ; set start address
                movwf FSR
clear           clrfs INDF               ; clear the RAM location
                incf FSR                 ; set for next location
                movlw RAMBlock + 0Ah     ; test if finished
                xorwf FSR,W
                btfss STATUS,Z
                goto clear               ; not yet, keep going
```

Knowing tricks like this becomes important when you start to write reasonably complicated code, because it lowers the chance of making mistakes, makes your code easier to understand, and makes it easier for you to write code.

There are lots of compiler directives in MPASM so don't be afraid to look through the help file supplied with MPLAB to check them out.

Open [MPLAB](#) and the file called `block.asm`. Click on `Options - Development mode` and set the processor to **16F84** then make sure the `MPLAB Sim - simulator` item is checked and then press `Reset` to close the window.

One other time saving directive you can use is the `INCLUDE` statement which you will see at the top of the code listing. Do you notice that the `FSR`, `STATUS`, `INDF` and the `Z` flag are not defined anywhere?

The `INCLUDE "P16f84.inc"` statement tells the assembler to find the file called `p16f84.inc` and assemble it as well as this code file. Inside this file are all the register and bit definitions for the PIC16F84 chip. This saves us having to type them out each time we write code for the 16F84 chip. These files are located in the same directory where MPLAB is located. They are just text files so you can view the contents of them by using NotePad.

You may notice that the Mnemonics have been written in CAPITALS. That is the way they are defined in the `INCLUDE` files, and the source files will not compile properly if the labels don't match exactly. There is a Case Sensitivity option that you can disable in the dialog box that appears when you assemble each file.

OK. Now type `ALT F10` and press `OK` to assemble the code.

Click on `Window - File Registers` to display the list of RAM locations. You may need to size the window so that you can easily see RAM address 0020 to 0060. Click on `Window - Special Function Registers` and size the window so that you can easily see the `FSR` register.

Now single step the code and watch how the code executes. You will see each RAM location change to **RED** as the value 00h as is written to it by Indirect Addressing. When this has completed, you will see the value 01h written to each location and then the process continues. Notice that the `FSR` value equals the value of the RAM register being written to.

What happens if you changed this line of code

```
xorwf FSR,W
```

to this

```
xorwf FSR
```

Try it and see. Did the code work properly?

Try to find out why. As a hint, where did the result of the `xorwf` instruction end up and what important value got changed as a consequence?.

Change the values of the size and start address of the RAM block and see what difference it makes.

Try writing or reading a value directly to RAM address 0h, which is the INDF register. I'll bet you can't.

You can read or write data to any RAM address in the 16F84 when you are using indirect addressing. That is because the FSR can hold address values from 0 - 255 which covers the whole address space.

Tables At The Boundary

Earlier on we looked at using the `ADDWF PCL` instruction to access different values in a lookup table, and this was the general format we used.

```
DataTable addwf PCL          ; add W value to PC
    retlw b'00111111'        ; 0
    retlw b'00000110'        ; 1
    retlw b'01011011'        ; 2
    retlw b'01001111'        ; 3
    retlw b'01100110'        ; 4
    retlw b'01101101'        ; 5
    retlw b'01111101'        ; 6
    retlw b'00000111'        ; 7
    retlw b'01111111'        ; 8
    retlw b'01101111'        ; 9
```

This method worked very well for the previous programs you looked at and will do well for most of the software you write. Unfortunately, this method does not always work, and in this section we will find out why.

Start [MPLAB](#), load the program called `pages.asm`, and assemble it.(Alt F10) This software is very similar to the program that displays the values 0 - 9 on a 7 segment display. The only difference in the code is where the `DataTable` subroutine is located in ROM, and there is no `Delay` subroutine included.

Begin stepping through the code and see if all of the digit information gets sent to `PORTB` for the display.

What happened when the program tried to get the information for digit 6 from the DataTable subroutine?

The program should have started from the first code line again which was wrong. This is a disaster, because this is not what we wanted and our program just crashed.

The trick now is to find out why.

Here is some of the list file code that was generated after you assembled pages.asm. You can view this entire file by loading pages.lst into MPLAB.

```
0000          00020          org 0h

0000 1683  00022          bsf STATUS,RP0
0001 0185  00023          clrf TRISA
0002 0186  00024          clrf TRISB
0003 1283  00025          bcf STATUS,RP0
0004 01A0  00027  Loop1   clrf Pointer
0005 0820  00028  Loop2   movf Pointer,w
0006 20F9  00029          call DataTable
0007 0086  00030          movwf PORTB
0008 0AA0  00031          incf Pointer
0009 300A  00032          movlw d'10'
000A 0620  00033          xorwf Pointer,w
000B 1D03  00034          btfss STATUS,Z
000C 2805  00035          goto Loop2
000D 2804  00036          goto Loop1

00F9          00038          org 0x00F9

00F9 0782  00053  DataTable addwf PCL
00FA 343F  00054          retlw b'00111111' ; 0
00FB 3406  00055          retlw b'00000110' ; 1
00FC 345B  00056          retlw b'01011011' ; 2
00FD 344F  00057          retlw b'01001111' ; 3
00FE 3466  00058          retlw b'01100110' ; 4
00FF 346D  00059          retlw b'01101101' ; 5
0100 347D  00060          retlw b'01111101' ; 6
0101 3407  00061          retlw b'00000111' ; 7
0102 347F  00062          retlw b'01111111' ; 8
0103 346F  00063          retlw b'01101111' ; 9
```

The list file is divided into columns. The first shows the ROM address that this line of code is assigned when the chip is programmed. The second shows the HEX code that was generated from the source on this line. Then there is the text line number and finally the source code you wrote.

From this listing you can clearly see how the `ORG` statements change the assemblers address counter. For example, the `ORG 0x00F9` changes the address counter to 00F9h and this new ROM address is where the `DataTable` subroutine code will start from when it is programmed into the chip.

So far so good?

If you are not sure how this process works, start the [MicroPro](#) programmer software and select the 16F84 chip.

Now load the `pages.hex` file that you just created by assembling `pages.asm`, and you should see that each HEX code value is placed into the programmer listing at the same addresses that appear in the `pages.lst` file.

Scroll down the HEX listing and you will see the code for the `DataTable` subroutine and how it begins at address 00F9h. Here, it should have the HEX code value of 0782 which is the value that the assembler created when it read this source code line.

```
DataTable addwf PCL          ; add W value to PC
```

OK then. Now that you understand that process, why didn't the program work?

The answer lies in the fact that the PIC is an 8 bit device. Therefore the `PCL` register and all the others can only hold values from 0 - 255. (00h - FFh)

Hang onto your seats here, because we have to do a bit of math.

Look at the highlighted ROM addresses in the `DataTable` code.

```
00F9 0782 00053   DataTable  addwf PCL
00FA 343F 00054           retlw b'00111111'   ; 0
00FB 3406 00055           retlw b'00000110'   ; 1
00FC 345B 00056           retlw b'01011011'   ; 2
00FD 344F 00057           retlw b'01001111'   ; 3
00FE 3466 00058           retlw b'01100110'   ; 4
00FF 346D 00059           retlw b'01101101'   ; 5
0100 347D 00060           retlw b'01111101'   ; 6
0101 3407 00061           retlw b'00000111'   ; 7
0102 347F 00062           retlw b'01111111'   ; 8
0103 346F 00063           retlw b'01101111'   ; 9
```

The `PCL` register is always the same the lower 8 bit value of the Program Counter which means that it equals **F9h** when the subroutine starts. The `W` register value is added to the `PCL` value, and this changes the value of the program counter. This, as we have seen before, provides a neat way of retrieving data from a lookup table.

Let's look again at how the digit 5 data is accessed.

PCL = F9h W = 5h

As the PIC is executing the `addwf PCL` instruction, it increments the Program Counter so it can start fetching the next instruction.

New PCL value FAh

Now the W register value is added to the PCL value.

New PCL value FAh + 5h = FFh (255dec)

As the lower 8 bits of the Program Counter always equal the PCL value, it now equals **00FFh**.

The `retlw b'01101101'` instruction now executes and returns the 7 segment data for digit 5.

Now, what happens when we want to display digit 6.

PCL = F9h W = 6h

As the PIC is executing the `addwf PCL` instruction, it increments the Program Counter so it can start fetching the next instruction.

New PCL value FAh

Now the W register value is added to the PCL value.

New PCL value FAh + 6h = 00h (0dec)

As the lower 8 bits of the Program Counter always equal the PCL value, it now equals 0000h.

Now this instruction executes.

```
0000 1683 00022                      bsf STATUS,RP0
```

Whoa there!! Hang on a minute. That's not right.

Well yes, it is correct, but the PIC didn't do anything wrong and it followed the code we wrote to the letter. Unfortunately, we made a mistake in the way we wrote the code.

You can see that the PCL value “wrapped around” back to zero after the W register value was added. This happened because you cannot store a value in a register that is greater than 8 bits.

$$\text{FAh} + 6\text{h} = 100\text{h} \quad (250 + 6 = 256)$$

That is a nine bit number 1 0000 0000

As the result of the addition can only be 8 bits long, the extra ‘1’ disappears and the final result equals 0000 0000, or zero.

The same happens for larger numbers.

$$\text{FFh} + \text{FFh} = 1\text{FE} \quad (510\text{dec})$$

That is a nine bit number 1 1111 1110

As the result of the addition can only be 8 bits long, the extra ‘1’ disappears and the final result equals 1111 1110, or FEh. (254dec)

When the subroutine tried to get the 7 segment data for digit 6, the Program Counter was incorrectly set to 0000h, which is back at the start of the program. Clearly, this is an error.

If you have not understood what has been said up to this point, please go over it again before moving on.

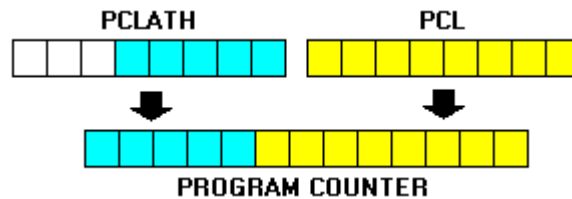
To overcome this problem, we can proceed two ways. One way is to make sure that any of our lookup tables do not appear in any ROM addresses greater than 00FFh. This may be good enough for a lot of programs, but sometimes it isn’t.

What if you have a lot of data tables, or just one really big one that has more than 256 elements listed. In these cases, you have to learn about another trick when using PIC chips.

We saw earlier that the PCLATH register is used to access different ROM pages when using the GOTO and CALL instructions. This register is also used when any instruction causes the PCL register contents to be changed.

```
addwf PCL
subwf PCL
incf PCL
etc
```

When the PCL register is changed, so too does the lower 8 bits of the Program Counter. In addition, the lower 5 bits of the PCLATH register are used as the new upper 5 bits of the Program Counter which gives us a new 13 bit value.



This may seem a little bit tricky, but it is quite simple and we can use this knowledge to create lookup tables that contain more than 255 elements, or ones that can be located anywhere in memory.

When the PIC powers up, some internal registers are set to a RESET value. The PCLATH is one of those registers and its value is set to zero when power is first applied. That is the reason why the `pages.asm` program worked with `Pointer` values less than 6.

Imagine, if the PCLATH was set to 1 on powerup.

Load the program called `pages2.asm` into MPLAB and assemble it.(ALT F10)

This program sets the PCLATH value to 1 before anything else. Single step the code and the `DataTable` subroutine will not work at all. The Program Counter will be set to an address where there is no code and an error window will be displayed.

The way around this is to set the PCLATH register in such a way that the Program Counter is always set to the correct address when the PCL register is changed.

Using the `HIGH` and `LOW` compiler directives is one way this can be achieved. You should remember that this gets the upper or lower 8 bits of any 16 bit value.

We want to make sure digit 6 works, so let's use that as an example.

Use the `HIGH` directive to get the upper 8 bits of the ROM address where the `DataTable` subroutine is located.

```
movlw High(DataTable)
```

Here, the assembler finds out where the `DataTable` subroutine ends up in ROM memory and makes sure the W register is loaded with the upper 8 bits of this value. The routine is located at address `00F9h`, so code is generated to make sure the W register is set to `00h`. This value is placed into the PCLATH register.

Now, get the table offset, in this case it is the `Pointer` value, and add 1 to it.

We now add this to the lower 8 bits of the ROM address where the `DataTable` subroutine is located.

```
addlw Low(DataTable)
```

The assembler finds out where the `DataTable` subroutine ends up in ROM memory. The routine is located at address `00F9h`, so code is generated to make sure the W register is added to the value `F9h`.

After we do that, the CARRY flag in the STATUS register will be set to Logic 1 if the value exceeds 255, or it will be cleared to Logic 0 if the value does not exceed 255. If the CARRY flag is set after the addition, we increment the contents of the PCLATH register.

After doing this, the PCLATH will be set to the correct value to access the data element we are looking for. Also, data tables can be placed anywhere in memory and the PCLATH will be set correctly each time.

One limitation of this method, is that you are still limited to tables which have 254 elements in them. If you had more, then you need to do some 16 bit arithmetic to calculate the PCLATH value which isn't covered in this topic. (It's still easy though.)

Let's do the math for the above example.

The `DataTable` subroutine is located at address `00F9h` so W is set to `00h` by the assembler. This is then placed into the PCLATH register.

PCLATH = 0

We now get the offset (`Pointer`) value and add 1 to it.

$6 + 1 = 7$

This value is now added to the lower 8 bits of the `DataTable` subroutine address.

$$7 + F9h = 100h$$

The result is greater than FFh (255) so the CARRY flag is set. This means we have to increment the PCLATH by 1.

$$PCLATH = 1$$

The PCLATH is now set to the correct value and the data for digit 6 will be returned properly.

Here is the code to set the PCLATH correctly.

```
movlw High(DataTable)
movwf PCLATH
incf Pointer,W
addlw Low(DataTable)
btfsc STATUS,C
incf PCLATH
```

You can think of setting the PCLATH value in these terms also if you like.

If the PCLATH = 0, addwf PCL accesses memory 0h - FFh.

If the PCLATH = 1, addwf PCL accesses memory 100h - 1FFh.

If the PCLATH = 2, addwf PCL accesses memory 200h - 2FFh.

If the PCLATH = 3, addwf PCL accesses memory 300h - 3FFh.

etc.

It's not just ADDWF instructions, it's any instruction that changes the PCL.

Load the file called `pages3.asm` into MPLAB and assemble it. Now load `pages3.lst` and verify that `movlw High(DataTable)` and `addlw Low(DataTable)` actually do generate the correct code. The lower 8 bits of the HEX code will be the value used.

```
0007 3000 00031 Loop2      movlw High(DataTable)
000A 3EF9 00034           addlw Low(DataTable)
00F9 0782 00063 DataTable  addwf PCL
```

Now single step through the code and verify that it now works. Click on Window - Special Function Registers to view the PCLATH.

Try changing the `org 0x00F9` line to different values and make sure that the code still works. You will see the different values calculated by the `High` and `Low` directives in the new list files that are created.

Try `org 0x0345h`

Try `0x03FEh`. You will find that the code compiles ok, but you get warnings like this if the assembler is set to compile for a 16F84.

```
Warning[220] C:\...\PAGES3.ASM 65 : Address exceeds maximum
range for this processor.
```

That is because the assembler is trying to fit code into addresses that do not exist for the 16F84.

There are some animated tutorials covering this subject in the [MicroPost](#) program under the headings **Modify PCL** and **GOTO CALL PCL**.

Using The EEPROM

The EEPROM memory is different than normal RAM memory because the data that is stored there stays intact even when power is removed. This can be very helpful when you want to keep special data for your program. As an example, the EEPROM data could be used for the odometer information on a speedometer, or it could be a value that your program uses for calibration purposes.

Using the EEPROM memory is very easy but you must always use a special sequence of instructions to read or write to it.

This section deals with reading and writing to the EEPROM memory on a PIC16F84.

There are four registers used to enable you to access the EEPROM memory. These registers are called:

- EEDATA
- EEADR
- EECON1
- EECON2

The `EEDATA` register is used as a temporary register to hold the **data** when you access the EEPROM.

The `EEADR` register is used to hold the **address** of the EEPROM location that you want to read from or write to.

The `EECON1` register has all the **control bits** necessary to enable you to read and write to the EEPROM memory.

The `EECON2` register is a special register that enables you to write the data.

The easiest way to explain these registers is to use an example so here we go.

Reading

Reading is quite an easy task. Now you can't just jump straight in and say, hey I want to read from the EEPROM. As you might expect, you have to let the PIC know which memory location you have to read from because there are 64 of them available for storage.

The first thing you must do then is write the address of the location you wish to read from into the `EEADR` register. Say you want to read from the second location. In this case you can use these instructions.

```
movlw 1h
movwf EEADR
```

Hold on a minute. We were going to read the second EEPROM location, so why use 1h as the value? Remember, address 0, (0h) is the first location, and address 63, (3Fh) is the 64th location. All memory is zero based, which means it doesn't start from 1.

Here are the bits that live in the `EECON1` register which is at RAM address 88h.

`EECON1`

BIT NUMBER	7	6	5	4	3	2	1	0
FUNCTION	-	-	-	EEIF	WRERR	WREN	WR	RD

Now to tell the PIC that you actually want to read a value from EEPROM, you need to set one of the special control bits in the `EECON1` register. This special bit is called `RD`, (BIT 0) which is short for `READ`.

The `EECON1` register is located in RAM Page 1, so before accessing it, you need to set the `RP0` bit in the `STATUS` register.

```
bsf STATUS,RP0          ; RAM Page 1
```

To read the location, just set the `RD` bit to Logic 1.

```
bsf EECON1,RD           ; read the data
```

When you set the `RD` bit to Logic 1, the PIC automatically looks at the value stored in `EEADR` and uses it as the EEPROM memory address from which to read from. After it has read the location, it places the data into the `EEDATA` register, and when this has been completed the `RD` bit gets set back to Logic 0 again. This all happens during the `bsf EECON1,RD` instruction.

The data is now available in the `EEDATA` register and you can do what you want with it. This register is located in RAM page 0, so you must clear the `RP0` bit in the `STATUS` register before you access it.

```
bcf STATUS,RP0          ; RAM Page 0
movf EEDATA,W
```

Here is the full list of instructions needed to read from the EEPROM.

```
movlw 1h
movwf EEADR
bsf STATUS,RP0          ; RAM Page 1
bsf EECON1,RD           ; read the data
bcf STATUS,RP0          ; RAM Page 0
movf EEDATA,W
```

You can make this procedure into a SUBROUTINE quite easily, and all you have to do to use it is set the `W` register with the EEPROM address and after the subroutine executes, the data is returned in the `W` register.

```
;
; SUBROUTINE: READ FROM EEPROM
; On entry, W = address to read from
; On exit, W = data that was read
;
EepRead  movwf EEADR
        bsf STATUS,RP0          ; RAM Page 1
        bsf EECON1,RD           ; read the data
        bcf STATUS,RP0          ; RAM Page 0
        movf EEDATA,W
        return
```

To use this subroutine just set the W register with the address data and call it. Suppose you want to read all of the EEPROM and write the data to PORTB. Here is a way you could do that.

```
EEloop    clr  EEADR          ; set for first address
          call EepRead       ; get the data
          movwf PORTB        ; send to PORTB
          incf EEADR         ; set for next address
          movlw d'64'        ; test for last address
          xorwf EEADR,W
          btfss STATUS,Z
          goto EEloop        ; not last

; all done - continue with other code
```

The EepRead routine has the opportunity to fail if you are not careful. Can you see a reason for this?

What happens if the RP0 bit in STATUS is set to 1 when the subroutine is called? The EEADR register will not be accessed, and the EECON2 register will be.

In this code we are using both RAM pages, so you must always make sure you are in the correct one.

If you cannot be sure what RAM page you will be in when the EepRead subroutine is called, clear the RP0 bit inside the subroutine before doing anything else, like this.

```
;
; SUBROUTINE: READ FROM EEPROM
; On entry, W = address to read from
; On exit, W = data that was read
;
EepRead    bcf STATUS,RP0          ; RAM Page 0
          movwf EEADR
          bsf STATUS,RP0          ; RAM Page 1
          bsf EECON1,RD           ; read the data
          bcf STATUS,RP0          ; RAM Page 0
          movf EEDATA,W
          return
```

This is a really good example of seeing the RP0 bit in action. You can see how many times it was used and this routine would definitely not work if you ignored it.

Writing

Writing to the EEPROM memory is a little bit more difficult. You must write a special sequence of instructions every time you want to write data, and if you fail to do this, you will not be able to store the data at all.

The first thing you must do is set the address of the location you wish to write to into the `EEADR` register, and you must also set the value of the data that is going to be stored into the `EEDATA` register. Say you want to write `0x45` to the first location. In this case you can use these instructions.

```
movlw 0h
movwf EEADR
movlw 0x45
movwf EEDATA
```

There is no strict order for setting these registers.

Now to tell the PIC that you actually want to write a value to EEPROM, you need to set some of the special control bits in the `EECON1` register.

One of these special bit is called `WT`, (BIT 1) which is short for `WRITE`, and the other is called `WREN`, (Bit 2) which is short for `WRITE ENABLE`.

These bits sound much the same, but have quite different functions.

Here are the bits that live in the `EECON1` register which is at RAM address `88h`.

`EECON1`

BIT NUMBER	7	6	5	4	3	2	1	0
FUNCTION	-	-	-	EEIF	WRERR	WREN	WR	RD

Before writing can take place, the `WREN` bit must be set to Logic 1. If this bit is Logic 0, then the PIC will not write any data even if it has been told to.

The `EECON1` register is located in RAM Page 1, so before accessing it, you need to set the `RP0` bit in the `STATUS` register.

```
bsf STATUS,RP0           ; RAM Page 1
```

To enable writing, just set the `WREN` bit to Logic 1.

```
bsf EECON1,WREN          ; enable EEPROM writes
```

To make the PIC start writing data to the EEPROM, you **MUST** use these instructions, and they **MUST** be written exactly like this and in the same order.

In conjunction with the `WREN` bit, this sequence is a safe guard against erroneous writing. The `EECON2` is also in RAM page 1 and the `RP0` bit is already set previously so there is no need to set it again.

```
movlw 0x55          ; begin writing
movwf EECON2
movlw 0xAA
movwf EECON2
bsf EECON1,WT
```

After these instructions execute and you set the `WT` bit to Logic 1, the PIC automatically looks at the value stored in `EEADR` and uses it as the EEPROM memory address from which to write to. It also looks at the value stored in `EEDATA` and uses this as the data that will be written.

The PIC will take some time to write this new data because that is the nature of EEPROM memory. You can read from it at the same speed as normal RAM, but it takes a lot longer to store data which, as you know, will be permanent until overwritten again.

After the PIC has finished writing the data to the specified location, the `WT` bit is automatically set back to Logic 0 again and the `EEIF` flag gets set to Logic 1. The time it takes for the location to be written to varies a little bit, but is usually around 10 milli-seconds. (10mS)

This doesn't sound like much, but it is quite a long time in microprocessor terms. A PIC running with a clock speed of 4MHz can process 10,000 instructions during this write time.

You cannot write another byte of data to the EEPROM unless the PIC has finished writing that last byte. If you try, you will more than likely get errors.

There are two ways to tell when the write cycle is completed.

One is by testing the `WR` bit for a Logic 1, and the other is by using interrupts. We will not discuss using interrupts here.

After the sequence of instructions is completed, the last instruction sets the `WR` bit to Logic 1. The data is now written to the EEPROM and when finished the `WR` bit is automatically cleared back to Logic 0. We can use these instructions to create a loop to wait around and test for this condition. The `RP0` bit is still set for RAM page 1, so there is no need to worry about it.

```
WriteWait btfsc EECON1,WR      ; wait for write completion
          goto WriteWait
```

Once this loop exits, it is safe to write another byte of data.

If the write cycle is started but gets interrupted by something like a MCLR or WDT reset, then the WRERR bit in EECON1 is set to Logic 1. This is the WRITE ERROR bit. If you do a write to EEPROM write and afterwards this bit equals Logic 1, then you will have to rewrite the location.

Here is the full list of instructions in a subroutine.

```

;
; SUBROUTINE: WRITE TO EEPROM
; On entry, W = data to be written
; and EEADR is already set
;
EepWrite  movwf EEDATA
          bsf STATUS,RP0          ; RAM Page 1
          bsf EECON1,WREN         ; enable EEPROM writes
          movlw 0x55              ; begin writing
          movwf EECON2
          movlw 0xAA
          movwf EECON2
          bsf EECON1,WR
WriteWait btfsc EECON1,WR         ; wait for write completion
          goto WriteWait
          bcf STATUS,RP0          ; RAM Page 0
          return

```

To use the EEPROM write subroutine, you must write the data to the EEADR register and then load the W register with the data that will be written.

In this example, PORTB will be read 64 times and the data stored in each consecutive EEPROM location. The EEADR register always has the correct address value before calling the EepWrite subroutine.

```

          clrf EEADR              ; set first EEPROM address
EepLoop   movf PORTB,W            ; get data from PORTB
          call EepWrite           ; write the data
          incf EEADR              ; next address location
          movlw d'64'             ; test if written to all
          xorwf EEADR,W
          btfss STATUS,Z
          goto EepLoop           ; no

; all done

```

These examples are illustrated in the file called `eeptest.asm`. You can use [MPLAB](#) to assemble it and experiment with the code if you like.

If you try to write to an EEPROM address greater than 63dec, then the value will be truncated. For example, location 64 will goto address 0, location 65 to address 1, etc.

Using the 16F873 chip.

This section deals with reading and writing to the EEPROM memory on a PIC16F873 which is used on the MicroPro PCB. It is important for you to understand the differences with the EEPROM read/write routines with this chip because it is used with some of the software in **MICRO's**.

The code is essentially the same except that the EEPROM registers, EEDATA, EEADR, EECON1 and EECON2 are in different RAM pages. The 16F873 also has the ability to write data to it's own internal code memory (ROM) so there is an extra bit in the EECON1 register to handle this. This bit is called EEPGD (bit 7).

The MicroSim and MicroPlay simulators cannot simulate reading and writing to the EEPROM while the code is being simulated on the MicroPro board at the same time. This is because the simulators are set up for the 16F84 and the MicroPro PCB is using a 16F873 chip.

You can use the 16F84 EEPROM read/write routines on the MicroSim and MicroPlay simulators to see how they work when there are not used with the external MicroPro board.

Here is the code that is necessary to Read/Write from the 16F873 EEPROM.

```
;
; SUBROUTINE: READ FROM 16F873 EEPROM
; On entry, W = address to read from
; On exit, W = data that was read
;
EepRead    bcf STATUS,RP0           ; RAM Page 2
           bsf STATUS,RP1
           movwf EEADR
           bsf STATUS,RP0           ; RAM Page 3
           bcf EECON1,EEPGD         ; data EEPROM
           bsf EECON1,RD            ; read the data
           bcf STATUS,RP0           ; RAM Page 2
           movf EEDATA,W
           bcf STATUS,RP1           ; RAM Page 1
           return
;
; SUBROUTINE: WRITE TO 16F873 EEPROM
; On entry, W = data to be written
; and EEADR is already set
;
EepWrite    bsf STATUS,RP1           ; RAM Page 2
           bcf STATUS,RP0
           movwf EEDATA
           bsf STATUS,RP0           ; RAM Page 3
           bcf EECON1,EEPGD         ; data EEPROM
           bsf EECON1,WREN          ; enable EEPROM writes
           movlw 0x55               ; begin writing
```

```

        movwf EECON2
        movlw 0xAA
        movwf EECON2
        bsf EECON1,WR
WriteWait btfsc EECON1,WR          ; wait for write completion
        goto WriteWait
        bcf STATUS,RP1           ; RAM Page 0
        bcf STATUS,RP0
        return

```

A similar READ example can be used with the 16F873.

```

        bsf STATUS,RP1           ; RAM Page 2
        clrf EEADR               ; set for first address
EEloop  call EepRead              ; get the data
        movwf PORTB              ; send to PORTB
        bsf STATUS,RP1           ; RAM Page 2
        incf EEADR                ; set for next address
        movlw d'64'              ; test for last address
        xorwf EEADR,W
        btfss STATUS,Z
        goto EEloop              ; not last
        bcf STATUS,RP1           ; RAM Page 0

```

Here is a similar routine for writing.

```

        bsf STATUS,RP1           ; RAM Page 2
        clrf EEADR               ; set first EEPROM address
EepLoop bcf STATUS,RP1           ; RAM Page 0
        movf PORTB,W              ; get data from PORTB
        call EepWrite             ; write the data
        bsf STATUS,RP1           ; RAM Page 2
        incf EEADR                ; next address location
        movlw d'64'              ; test if written to all
        xorwf EEADR,W
        btfss STATUS,Z
        goto EepLoop             ; no
        bcf STATUS,RP1           ; RAM Page 0

```

As you can see, the routines are a little bit more difficult because of the extra RAM pages that are used in the 16F873.

KeyPads

Sometimes in your projects you need to be able to get input data from a user. This may be as simple as a switch operating, or as complex as a serial interface from a PC or other types of equipment.

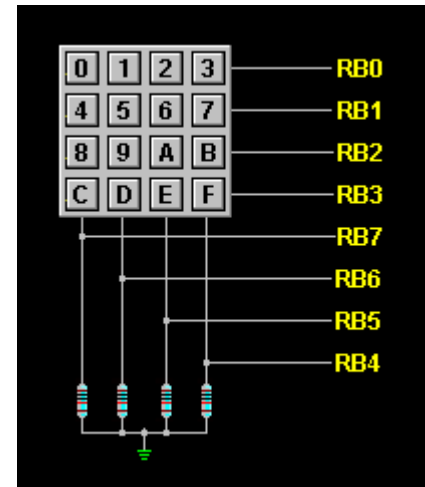
A common input device is the keypad, and for hobby use, these usually consist of 12 or 16 keys arranged as 3 X 4 or 4 X 4 respectively.

In this example we will look at the 16 key variety.

These keypads are arranged in a 4 X 4 format as shown here. The switch connections are wired in a matrix type of arrangement to keep the connector count as low as possible.

You can see in the diagram that the keys are arranged in 4 Rows and 4 columns. Port pins RB0 to RB3 are connected to the Rows and pins RB4 - RB7 are connected to the Columns.

The operation of these keypads is quite easy as each key is just a simple switch.



If key '0' is pressed, this creates a short between Row 1 and Column 1

If key '1' is pressed, this creates a short between Row 1 and Column 2

If key '2' is pressed, this creates a short between Row 1 and Column 3

If key '3' is pressed, this creates a short between Row 1 and Column 4

If key '4' is pressed, this creates a short between Row 2 and Column 1

If key 'F' is pressed, this creates a short between Row 4 and Column 4

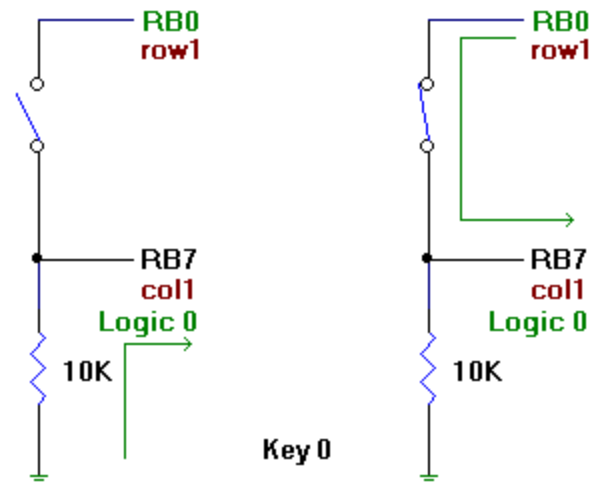
The task at hand is how to read this keypad with a PIC.

The solution is reasonably simple and involves writing software that will be able to detect if any key is pressed. This action can be called 'scanning a keypad'.

What if we set the PortB pins connected to the columns as inputs, and tied all of these to ground via 10K resistors as shown in the diagram. If we now read from PortB, we would see that RB4 to RB7 would all be at Logic 0.

Now suppose PortB pins RB0 to RB3 are set as outputs and are all at Logic 0.

In this state, would any of the input pins read any different if a key is pressed?
 No, they wouldn't. With no keys pressed, you would always read Logic 0's, and with any of the key pressed, you would still read Logic 0's.



What happens if we set pin RB0 as an output at Logic 1. That will put 5 volts onto the keypad along Row 1.

With no keys pressed, pins RB4 to RB7 will still read as Logic 0.

Now, suppose key '0' is pressed. That will cause pin RB7 to read as Logic 1.

If Key '1' is pressed, RB6 will read as Logic 1.

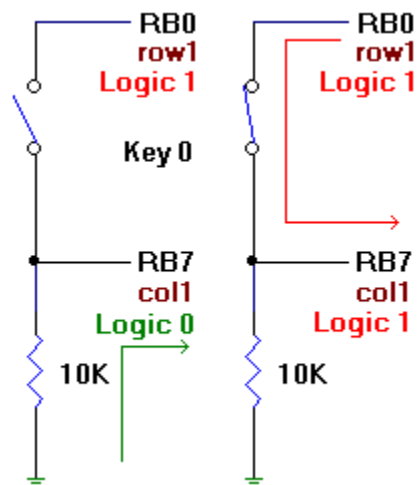
If Key '2' is pressed, RB5 will read as Logic 1.

If Key '3' is pressed, RB4 will read as Logic 1.

If Key '4' is pressed, RB7 will read as Logic 0.

Why didn't RB7 read a Logic 1 when Key '4' was pressed?

That is because we only set RB0 at Logic 1 and RB1 is still Logic 0, therefore Row 2 is Logic 0. When Key '4' is pressed you are still switching a Logic 0 level to RB7.



What you should have noticed is that when a row is set to 5 volts and a key is pressed on that same row, we can read the logic level change on the corresponding column input.

So all we have to do to scan the keypad, is to sequentially set each row to a Logic 1 and look at the columns to see if any are Logic 1.
If a column is Logic 1 then a key is pressed. If all columns are Logic 0 then no keys are pressed.

The first thing to do is decide how often, and how fast, to scan the keypad. The time may vary depending on your project requirements, but in this example, we will scan the keypad every 65 milli-seconds, or about 15 times per second.

In earlier projects we used a simple delay subroutine to slow things down while switches were read. In this example we will use a timer which is part of the PIC. It is called Timer 0, or TMR0, and is located at RAM address 1h. We can use this timer to do all sorts of timing and counting tasks depending on how it is set up.

If you want to get familiar with TMR0 see the tutorials presented in [MicroPrac](#).

Here is a simple program that demonstrates how TMR0 can be used to control how often a program loops. In this example it is about 65mS if the clock speed for the PIC is 4MHz.

```
org 0000h

bsf STATUS,RP0           ; ram page 1
movlw b'11000111'        ; set TMR0 internal clk
movwf OPTION_REG         ; prescale 1:256
bcf STATUS,RP0           ; ram page 0

MnLoop    btfss INTCON,T0IF      ; wait for TMR0 overflow
           goto MnLoop

           bcf INTCON,T0IF        ; clear TMR0 overflow flag

           goto MnLoop           ; continue forever
```

The register that controls how TMR0 operates is called the OPTION register and this is located in RAM page 1 at address 81h. If you look at page 16 of the 16F84 data sheet you can see some information about this register. It is located on the CD ROM in the PDF directory - 'pic16f8x.pdf'.

The first instructions set TMR0 so that is incremented from the PIC's internal clock (bit 5), the Prescaler is assigned to TMR0 (bit 3), and the Prescaler is set to divide by 256 (bits 2 - 0).

TMR0 is an 8 bit register so it can only hold values from 00h to FFh. If the value in TMR0 is FFh and it increments, the new value will be 00h. This is called overflowing, or wrapping around to zero.

Every time this event happens, a special bit in the INTCON register is automatically set to Logic 1. This bit is called the T0IF bit (bit 2 - see page 17 of the data sheet).

If you look at the code above you can see that the T0IF bit is continually tested. If it equals zero, the loop stays at the `MnLoop` line. When the T0IF bit equals 1, this loop is broken.

The second loop does nothing but clear the T0IF flag and then return to `MnLoop` again where it waits for another overflow condition.

You must write code to clear the T0IF bit if you want to use it again. The PIC does not clear this flag, it only sets it.

Notice the term FLAG. This is a name for a bit that changes when a certain event occurs. In this case, the event is TMR0 overflowing and the T0IF bit flags it as such.

Open [MPLAB](#), load the file called 'keypad.asm', and assemble it. (ALT F10). Close the build window and click on Window - Stopwatch.

Move the cursor so that it is positioned over this code line.

```
bcf INTCON,T0IF          ; clear TMR0 overflow flag
```

Now click on it with the RIGHT mouse button, and a popup window will appear. Click on Break Point(s). The code line should now be red.

```
bcf INTCON,T0IF          ; clear TMR0 overflow flag
```

This is a break point and after running, the code will execute to this line and halt.

Press the Run button. When the code stops, check the time on the Stopwatch window. It will be around 65mS. Press the Zero button and then the Run button again. The loop execution time will always be around 65mS.

TMR0 is incrementing by one every time the PIC's internal clock gets incremented by 256. This timing is caused by the prescaler being active, which is set for a clock divide ratio of 256.

Change this code line.

```
movlw b'11000111'      ; set TMR0 internal clk
movwf OPTION_REG        ; prescale 1:256
```

to this

```
movlw b'11000000'      ; set TMR0 internal clk
movwf OPTION_REG        ; prescale 1:2
```

Reassemble and set the break point again.

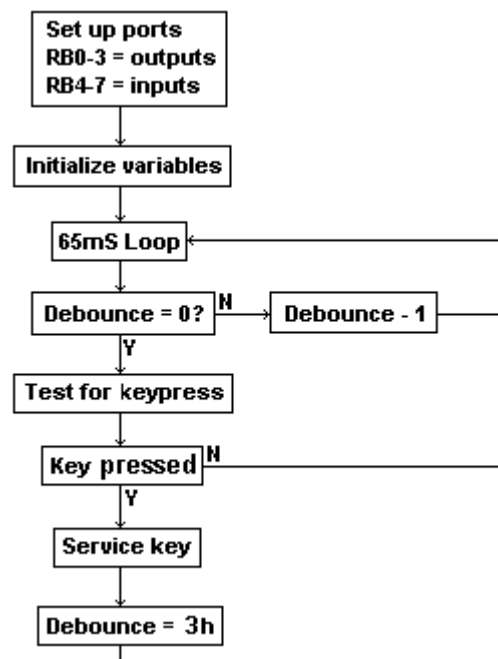
When you run the code you will notice that the loop time is considerably shorter - about 500uS. This is because you have just set the prescaler for a clock divide ratio of 2, which means TMR0 will now increment once for every 2 PIC clock cycles.

At this stage we have figured out how to make a code loop execute every 65mS, and now we need to use it to scan the keypad. Here is a flow chart of how the program will operate.

It should be fairly easy to follow.

If you remember back when we talked about switches, we had to use debounce software to avoid switch bounce which caused the PIC to 'see' multiple switch operations. The keypads are no different, as they have switch bounce as well.

There is a major problem with this flow chart. Can you spot it?



If there is a key pressed, it will be serviced every time the Debounce variable equals zero. This is not a desirable situation because you would have to remove your finger very quickly after pressing a key to avoid repetitions.

What you need to do is make the software know the state of the keys from previous loops.

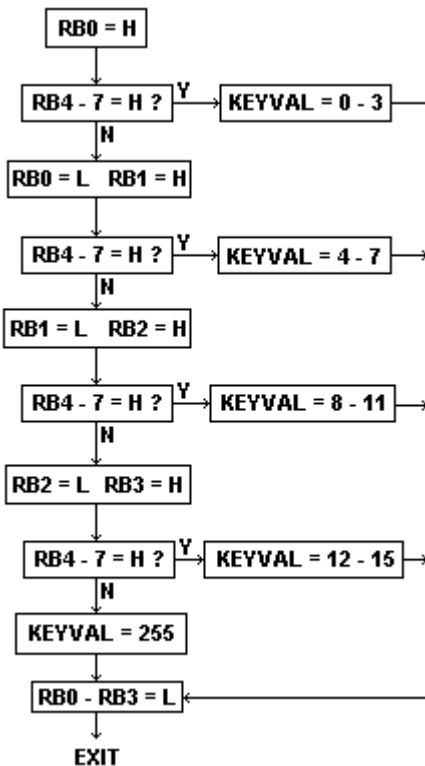
In this next flow chart you can see that the previous state of the keys is always known and if a key just changed state then you can take the appropriate action.

Take some time to go over this flow chart so that you understand how it works.

You will also notice that there is a **Test for keypress** box in the flow chart. This is the subroutine that does the actual scanning. After the keypad has been scanned, it would be nice to have the subroutine return with a value 0 - 15 which corresponds to a key being pressed, 0 - F. With a bit of thought it can be achieved.

Below is a flowchart that explains the keypad scanning routine.

What it does is make RB0 to RB3 go Logic 1 in turn. This sequentially puts 5 volts on each of the keypad rows.



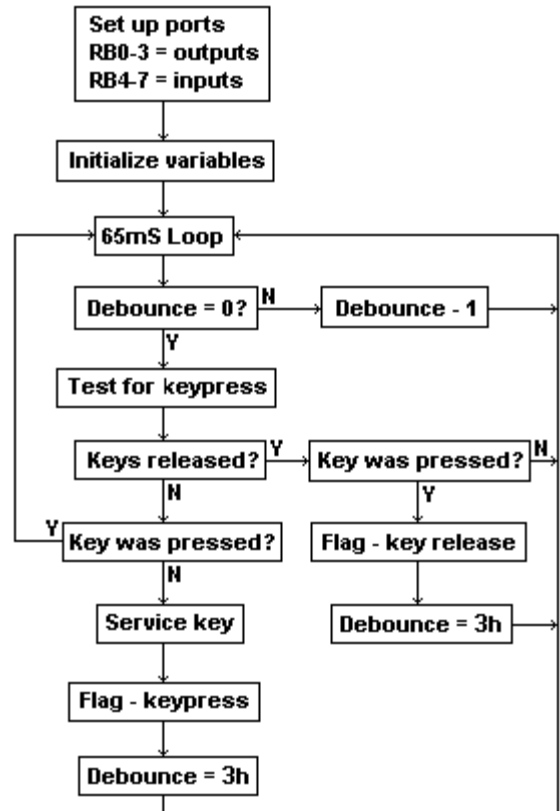
If any of the pins RB4 - RB7 are Logic 1 then a key has been pressed and the corresponding value (0 - 15) is returned in a variable called KEYVAL.

If there were no keys pressed the KeyCheck routine returns the value 255 (0xFF).

Rows will hold the value that is used to set each Row to Logic 1 in turn. Columns is a variable that holds the value returned when the column bits RB4 - RB7 are read. KeyVal is the value of the keypress or 0xFF if none pressed.

```

KeyCheck  movlw b'00000001' ; make ready to set Row 1 high (RB0)
          movwf Rows
          clrf KeyVal        ; initialise Key Value
  
```



These next lines set a row port bit to Logic 1 and then wait about 20uS for the port pins to stabilise before checking if any columns are Logic 1. If the software didn't wait a little bit, the PIC may read the column pins before they had time to change. Nothing in the real world changes instantaneously.

```
RowLoop    movf Rows,W           ; set a Row high
            movwf PORTB
            movlw 4h             ; small delay while PORTB stabilises
            movwf Count
Dloop      decfsz Count
            goto DLoop
```

Now that the pins have stabilised, check if any column pins are high, meaning a keypress. You can see how we only check bits RB4 - RB7 by masking the lower 4 bits with the AND instruction and then using the Z flag to test the result.

```
            movf PORTB,W         ; read portB
            andlw b'11110000'
            movwf Columns        ; save the data - only interested in upper 4 bits
            btfss STATUS,Z
            goto ChkColm        ; Z = clear, key must be pressed
```

After the AND instruction has executed, if any column bits were Logic 1, the Zero flag will be clear. If no keys are pressed and all columns are Logic 0, the Zero flag will be set.

Example when Row 1 = Logic 1.- no keys pressed

```
movf PORTB,W           ; read portB
```

```
W = 00000001
```

```
andlw b'11110000'
```

```
      0000 0001
```

```
AND   1111 0000
```

```
=      0000 0000
```

result is 0, **Z is set**

Example when Row 1 = Logic 1.- key '0' is pressed

```
movf PORTB,W          ; read portB
```

```
W = 10000001
```

```
andlw b'11110000'
```

```
      1000 0001
AND   1111 0000
```

```
=      1000 0000
```

result is not 0, Z is clear

If no keys are pressed, set the next Row to Logic 1. Also add, 4 to the KeyVal variable, because we are now looking at the next row of 4 keys.

```
movlw 4h          ; no bits set for this row
addwf KeyVal      ; looking at next row
goto NextRow
```

This next section of code is executed when a column bit was Logic one and the Z flag was clear. The idea is to check each of the 4 column bits and when we find one that is set, exit the routine and KeyVal equals the value of the keypress. If the bit we are testing is not set, then add 1 to the KeyVal variable and test the next bit. One of the 4 upper bits stored previously in Columns will be set.

```
ChkColm  bcf STATUS,C          ; clear the carry bit
          rlf Columns          ; if Carry = 1, key is pressed
          btfsc STATUS,C       ; key is pressed, KeyVal = key
          goto LowRow

          incf KeyVal          ; on next column so increment Key Value
          movf Columns        ; if = 0, then all columns checked for this row
          btfss STATUS,Z       ; check value of next column
          goto ChkColm
```

You can see that by using this method that KeyVal always equals the value of the current key being tested.

The current row that was tested did not have any keys pressed, so now we move on to the next. It is simply a matter of setting the current row to Logic 0 and the next to Logic 1 which can be done with the RLF instruction.

Rows = 0000 0001 for Row 1 - initial setting when routine started
Rows = 0000 0010 for Row 2
Rows = 0000 0100 for Row 3
Rows = 0000 1000 for Row 4
Rows = 0001 0000 when all rows tested

The value in Rows is written to PortB to set a keypad row to Logic 1. It is essential that you clear the carry bit before using the RLF Rows instruction. Can you guess why?

You can see also that when bit 4 of Rows = 1 that we have tested all of the rows on the keypad. If this happens then no keys have been pressed, so set KeyVal equal to 0xFF(255 dec) and exit.

```
NextRow    bcf STATUS,C           ; clear the carry bit
           rlf Rows              ; make ready to set next Row = high
           btfss Rows,4          ; test if all Rows have been set
           goto RowLoop         ; not yet, bit 4 will be set when all rows
checked
           movlw 0xFF            ; no keys pressed
           movwf KeyVal
```

This is where the routine exits so all keypad rows are set to Logic 0. At this point, if a key has been pressed, KeyVal will equal it's value (0 - 15), and if no key was pressed then KeyVal = 255.

```
LowRow     clrfs PORTB           ; all rows = low
           return               ; no key was pressed, KeyVal bit 7 still = 1
```

Please check this routine out thoroughly and be satisfied that you understand it before proceeding.

By the way, you had to clear the carry bit before using the RLF Rows instruction because you if you don't, and the carry bit is set, more than 1 keypad row will be set to Logic 1.

Here is the listing for the main code loop.

```
    clrf PORTA
    clrf PORTB
    bsf STATUS,RP0      ; ram page 1
    clrf TRISA          ; portA = all outputs
    movlw b'11110111'  ; PortB 7 - 4 = inputs
    movwf TRISB        ;      3 - 0 = outputs
    movlw b'11000010'  ; set TMR0 internal clk
    movwf OPTION_REG   ; prescale 1:256
    bcf STATUS,RP0     ; ram page 0

    clrf Debounce      ; these registers need to be
    clrf Flags         ; initialised to zero

MnLoop  btfss INTCON,T0IF ; wait for TMR0 overflow
        goto MnLoop     ; approx 65mS @ 4MHz clock

        bcf INTCON,T0IF

        movf Debounce    ; if Debounce = 0, test keypad
        btfsc STATUS,Z
        goto TestKP

        decf Debounce    ; else decrement it
        goto MnLoop     ; and ignore keypad
;
; TEST FOR A KEYPRESS ON THE KEYPAD
;
TestKP  call KeyCheck    ; check keypad

        btfss KeyVal,7   ; if = 1 then no key pressed
        goto KeyIsDown

        btfss Flags,Key  ; no key pressed
        goto MnLoop     ; and key is already released

        bcf Flags,Key    ; flag key is released
        movlw 3h         ; set debounce period
        movwf Debounce
        goto MnLoop

KeyIsDown btfsc Flags,Key ; if = 0 then key just pressed
        goto MnLoop     ; key is still pressed

        bsf Flags,Key    ; flag key is pressed
        movlw 3h         ; set debounce period
        movwf Debounce

        movf KeyVal,W    ; service key
        movwf PORTA     ; write KeyVal to PORTA

        goto MnLoop
```

You can notice in the loop that it knows when a key has been pressed or not by testing bit 7 of `KeyVal`. This bit will be 0 if a key is pressed because the value stored in `KeyVal` will only be from 0 - 15. This bit will be 1 if a key has not been pressed because the value stored in `KeyVal` will be 0xFF.

When a key has been pressed, and it has just been pressed, the value stored in `KeyVal` is written to `PORTA`. The value here can be viewed by connecting LEDs to the IO pins via a 1K resistor.

Each time a key is just pressed or just released, the `Debounce` counter is set to 3h. The keypad is not read again until this counter equals zero, thus we get a debounce delay of 3 loops or 3 X 65mS or 135mS.

The complete code listing is available in the Software directory and is called `keypad1.asm`. Load it into [MPLAB](#) and check it out.

In earlier tutorials we saw that we can add a value to the `PCL` to change the program counter. We can use that technique here to create a jump table to process each keypress and use `KeyVal` as the offset.

```
    movf KeyVal,W
    andlw 0fh          ; make sure KeyVal does not exceed 15
    addwf PCL
    goto Key1
    goto Key2
    goto Key3
    goto Key4
    goto Key5
    goto Key6
    goto Key7
    goto Key8
    goto Key9
    goto KeyA
    goto KeyB
    goto KeyC
    goto KeyD
    goto KeyE
    goto KeyF

Key1    ; process key 1
        goto MnLoop

etc
```

Be aware of ROM page boundaries if you use code like this.

Another problem which may arise when you create code loops with `TMR0`. To ensure even timing on every loop, you must make sure that all of the code inside the loop executes before `TMR0` overflows. In this application it may not matter much, but some of your applications may have critical timing requirements and will not function properly.