Kit 81

Simple PICmicro® Programmer/ Experimenter

DIY Electronics (HK) Ltd
PO Box 88458,
Sham Shui Po,
Hong Kong

http://www.kitsrus.com
mailto: peter@kitsrus.com

Last Modified 18 Nov 2002

DIY K81 Contents

Board Construction

The board is quite easy to construct as there are only a handful of parts.

It is advisable to read through the construction notes before starting.

WARNING

Be aware that the PIC16F84A chip is sensitive to static electricity discharge and could be damaged by mishandling. Do not touch the pins and only handle the chip by its ends.

Be careful with the board after assembly. Try to handle it only by the edges.

This project can work with the PIC16F84 or 16F84A chips.

Starting

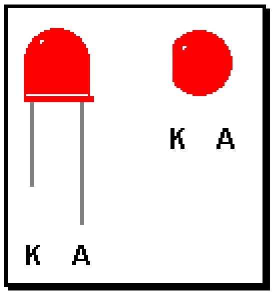
The first thing to do is inspect the PCB for shorted or open tracks or other damage. When you are satisfied that all is well, then you can proceed.

Start with the flattest parts first, which is the wire link near the parallel port connector. After that you can start placing the resistors. Hold each resistor body by the thumb and forefinger and use your other hand to loosely bend both of the leads over at right angles at the same time. Try not to make the bends too sharp, and you will find that they slide straight into the mounting holes on the board.

Leave the pigtails on each of the components until after soldering as the extra lead length serves as a heat sink for the component. Try not to leave the soldering iron on the components too long or you risk damaging them. The usual method is to hold the iron tip so that it touches the component lead and the PCB pad at the same time, and then apply a small dab of solder. This operation should only take about a second or two. If you are unsure of your soldering ability, find some spare components and practice on these before building the board.

Check that each solder joint is bright and shiny and doesn't look like a big dull blob which could mean a dry solder joint. The solder should flow freely onto the component lead and solder pad if it is to be a good joint.

After the resistors are soldered in, recheck your work and then mount the SIL resistors. The smaller of the 2 packages is the 10K, the larger is the 1K. Next mount the ceramic capacitors then the IC sockets.



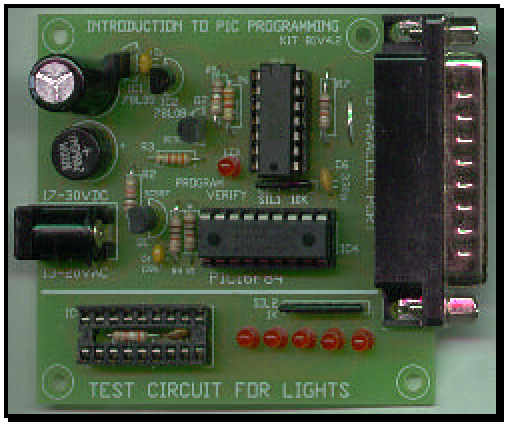
There are 6 LEDs that can be mounted next. Make sure they are oriented correctly. The Anode is marked on the PCB and is the longer of the two leads on the component. The Cathode has a flat surface on the LED body and is also marked on the board.

Next, mount the electrolytic capacitor C3. This component is polarity sensitive so make sure it is mounted properly. The positive lead is longer than the negative lead, and the negative lead is also marked on the side of the capacitor. On the PCB overlay, there are holes marked [+] as the positive lead for this component.

Now mount the BC557 transistors and then the 78L05 and 78L08 regulators making sure they are positioned correctly. These components all look the same so try not to get them mixed up. Then you can mount the bridge rectifier, the parallel port socket and the power jack.

That is all there is to the construction.

Now please go over your work and inspect it thoroughly. Check for missed or odd looking solder joints, bad component placement and orientation, and short circuits.



If you are satisfied that all is well, connect a suitable power source to the power jack. This can be 18VDC or around 13VAC.

Place a voltmeter with the negative lead on pin 7 of the 74LS07 chip socket and the positive lead on pin 14. The meter should read close to 5V. If it doesn' t, turn off the power and check the board again.

If all is well, turn off the power and insert the 74LS07 chip. Make sure it is placed into the socket correctly.

Connect a suitable parallel cable between the board and your PC.

Run the programmer software for the kit. **diyk81.exe**

Make sure you have the correct LPT port set. Click on `Port` to update it. If the port opens successfully then the buttons on screen will be enabled. If it failed to open then the buttons will not be enabled and you will have to try another port number.

Make sure the programming socket is empty.

Now press the Test Button. If all is well, the program will display...

```
Board test passed
```

If the software is not communicating with the board then it will display...

```
Board test failed
```

If it failed, then check the board for faults and check the parallel port and cable.

First Run

Make sure **diyk81.exe** is running and the parallel cable is connected to the project board. Apply power to the board.

Place a 16F84 chip into the programming socket making sure it is correctly positioned.

Press `Program`.

Select the file called `flash.hex` then press `OK`.

The file will be programmed into the chip and then verified.

If a problem occurred check that the chip is in the socket correctly, then check power, port and cable.

If it did program ok, turn off the power to the board and remove the chip from the socket. Now carefully place it into the LED demonstration socket and apply power again. You should see the LED closest to the port socket flash on and off.

Note: Please do not have the 16F84 chip in the programming socket unless the `diyk81.exe` program is running and has control of the circuit.

Now that the construction is complete, it’ s time to get into the best stuff.

Hold onto your hats and we will discover how the PIC code actually works.

Flash That LED

Flashing a LED 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 flies to the moon, 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” .

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. 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 problem inside the chip.

Remember:

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

One other thing that can cause problems is in the way you handle the chip. 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 don’ t 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. Handle the chips only by their ends.

The PIC16F84A data sheet is available in PDF format from the **Microchip** web site.

OK then, so how do we get started?

You might be tempted to jump right 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 well structured and documented, and in the future when you have forgotten what you wrote, you can read through and understand it more easily.

So just how do we get this piece of silicon to do our bidding, which in this case is to 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 and the software is a list of instructions that we create. The PIC will follow these instructions 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 for us by using a PIC. By this I mean the physical hardware needs of the project. You can’ t write reams of software without knowing what the PIC is going to control. With some projects you may find that you need to alter hardware and software as you progress through the development, 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 this current is about 20mA maximum. There is no real point in driving a LED with 20mA with this simple project, so let’ s drive it with 3mA to be on the safe side.

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 its pins.

These pins can be set as inputs or as outputs and they are sometimes referred to Input/Output (IO) pins.

When they are set as outputs we can make each individual pin provide 5 volts or 0 volts 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.

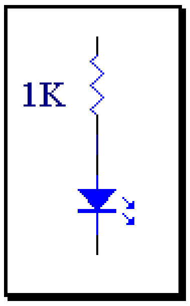
When these pins are configured as inputs we can read the value of the voltage level applied to them by some external circuit arrangement. If the pin has 5 volts applied, then the PIC will *see* a Logic 1. Conversely, if the pin has 0 volts applied then the PIC will *see* a Logic 0.

Previously, we said that we are going to control the LED with about 3mA of current which is well inside the PICs current rating. A red LED will consume about 2.4 volts across it when it is being used. We know that an output pin will supply 5 volts, so that means the series resistor needs to consume the remaining 2.6 volts. 5V - 2.4V = 2.6V. By using *Ohms Law* we can calculate the value of the resistor which must drop 2.6 volts with 3mA of current flowing through it and the LED.

V = I R or R = V / I R = 2.6 / 0.003

Therefore R = 866 ohms.

A general resistor around this value is 1000 ohms, so that is what we will use.



Therefore, 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 16F84A there are 13 pins available for us to use and these are divided into 2 Ports.

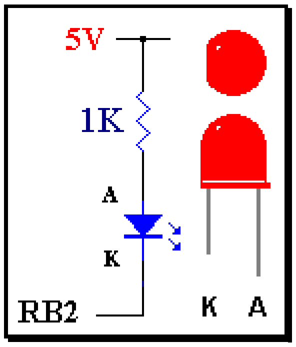
`PORTA` has up to 5 pins which are numbered `RA0 - RA4`.
`PORTB` has 8 pins which are numbered `RB0 - RB7`.

At this stage you might think that we can use any one of these, and you would be right. One thing to note however, `RA4` has an **open collector** output.

This means when this pin is set as an output it can only provide a connection 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 the 5 volt supply. In this project that fact doesn' t matter because all the LED anodes are connected to the 5V rail through a 1K resistor. This means the IO pins have to go *low* to light the LEDs.

There are a few hidden *gotcha's* that exist in the world of microcontrollers and the best way of knowing about them is by close examination of the device data sheets. 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.



Lets use pin RB2 to drive the LED in our first test program.

To summarise our project so far, we are going to flash a red LED in series with a 1K ohm resistor from PORTB pin RB2. As you may already know, a LED can only work if the anode is more positive than the cathode, so the cathode side of the LED will be connected to RB2 which will drive that end of the LED low.

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.

As you can now see, there' s not much point writing any software without knowing what the hardware will be.

Writing The Software

There are quite a few ways to create software for the PIC. You can write it with a simple text editor like *NotePad*, or use MPLAB from Microchip. There are other ways like using a BASIC or C complier.

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.

Over the page is the actual `flash.hex` code that was programmed into the PIC to make it flash the LED connected to pin RB2.

```
:1000000000308500FF308600831600308500003008
:1000100086008312FB3086001120FF3086001120FD
:100020000A288C018D0103308E008C0B15288D0B56
:0800300015288E0B15280800AD
:02400E00FB3F76
:00000001FF
```

It' s not very meaningful to us is it, but that does not matter because we are not computers.

Hex numbers may be new to you so it will be best to have a quick look at them now. At some stage you will 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 computer, 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 only have the numbers 1 and 0 to use, it would seem obvious that after we count 0 then 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.

Decimal	Binary
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. This table emphasises the bit count. Remember, 0 bits represent a logic 0 value so we should not leave them out.

Decimal	Binary
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.

Notice how letters A - F are used after the number 9

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*.

Decimal	Binary	Hex
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

If you look at the data sheet for the PIC 16F84A, 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.

Decimal	Binary	Hex
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

If you are dealing with PIC programming and are working with a PIC assembler then hex numbers that *begin with a letter* should be preceded by 0x.

0xFF45AC 0xA786097

You can also use this notation if the value starts with a number.

0x8CA08CA0 0x000011

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*.

Remember:

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

Well then, just how do we create a hex file like this.

The answer is - by using an *Assembler*.

The Assembler

Assembler is *easy - easy - easy*.

An assembler is quite an ingenious piece of software because it can read a text file that we have written as 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 especially in large projects.

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 16F84A has 1024 14 bit words available for program storage. In computer language 1024 means 1K, so this PIC has 1K of program space.

Assembler 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. This can be 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.

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*.

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 suitably 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 following 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  "Simple Program"

          list p=16f84A        ; processor type

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

start     movlw 0x04          ; simple code
          movwf 0x06
          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 `0x04`. 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 16F84A chip. This feature can be helpful if you write too much code for this particular processor to use.

The assembler will check how much code it has assembled and let you know if it is too much to fit in that particular chip. Directives like these control how the assembler builds the final object code.

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

```
;
; -------------
; PROGRAM START
; -------------
;
```

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. Notice how comments begin with a semi colon (;). Any line that begins with one of these is ignored.

`Org 0h` is another directive that tells the assembler to set the current memory address where the following instructions will be placed into. Remember that the 16F84A has 1024 location available for code use.`Org 0h` tells the assembler to start loading the instructions starting from memory address 0. In other words the `movlw 0x00` instruction will occupy memory location 0 and the `movwf 0x05` instruction will occupy memory location 1. `goto start` will occupy memory location 2.

One thing you may have missed here, is the fact that memory addresses start from location 0, not location 1. The code memory 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 code memory locations as ROM addresses - *Read Only Memory*.

`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 with the same name either. That confuses the assembler because it does not know which particular label you are referring to and 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.

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 0x04               ; simple code
          movwf 0x06
          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 at the `start` label again. This particular code will loop forever or until you remove power from the chip. Your code will *always* have some sort of loop even if you did not create one.

How is this so I hear you ask. Well, if you only write this code line...

```
start     movlw 0x04               ; simple code
```

...what happens after it is executed. The PIC doesn’ t stop just because you didn’ t write anything else. It will happily increment to the next code address and execute whatever is there - in this case nothing.

As a matter of fact nothing will be programmed into the rest of the ROM space either. They will all be blank and these will have the value `0x3FFF` in them.

The PIC processor actually decodes `0x3FFF` as `ADDLW 0xFF`. This means to add the value `0xFF` to the `W` register, or `ADD` Literal to `W`. So even though you only wrote one code line the PIC will execute `ADDLW 0xFF` 1023 times and then the program counter will wrap around back to `0x0000` and execute your code line again.

So what exactly does this small piece of code do?

```
start     movlw 0x04               ; simple code
          movwf 0x06
          goto start               ; do this loop forever
```

Remember our LED flash problem?

The LED was connected to pin `RB2` which is `PORTB` pin 2.

If we had to turn the LED on we have to write a Logic 1 to this pin. Remember that once we have done this, 5 volts will appear on the pin if it is set as an output.

What value should we write to `PORTB` if we want to set *only* pin `RB2` to Logic 1? Here is a small sample of a binary table.

Decimal	Binary
0	00000000
1	00000001
2	00000010
4	00000100
16	00010000
47	00101111

If we consider that pin `RB2` is represented by bit 2 in this table we can see that we need to write decimal value 4. Bit 0 is the far right binary bit, bit 7 is the far left.

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

In our case, to turn the LED on, we can write the value `0x04` to `PORTB`, and to turn the LED off we can write `0x00` to `PORTB`. So just how do we do this? If you look at the original code, this is the first line.

```
start     movlw 0x04              ; simple code
```

First we have a label called `start`, followed by the instruction `movlw 0x04`.

`movlw` means to **mov**e a **l**iteral value into the **W** register. A literal value is any value that can fit into 8 bits. Remembering that the PIC is an 8 bit device, this means a literal value can be any value from 0 to 255, (0x00 to 0xFF). In this case it is the value 4, or `0x04`. This instruction can also be written as

```
     movlw b'00000100'     ; binary notation
     movlw d'4'            ; decimal notation
     movlw 4h              ; another type of hex notation.
     movlw 04h             ; another type of hex notation.
     movlw 0x04            ; another type of hex notation.
```

Binary notation is quite good for writing to the ports because any bits in the value 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 because it is an *Open Collector* output.

Looking back at the code, the next line is

```
movwf 0x06
```

`movwf` means to **mov**e the contents of the **W** register to the **f**ile register specified. In this case it is RAM address `0x06`.

If you look at the 16F84A data sheet, and most other PICs for that matter, you will see that RAM address 6 is `PORTB`. So in other words, this instruction moves the contents of `W` to `PORTB`.

This seems a lot of effort. Why can' t we just write 1 to `PORTB`? 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*.

Mnemonics

This is a funny looking word. You pronounce it

```
Nem  On  Icks.
```

These items 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 file register [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 MPASMwin. This is a free program and is available from the Microchip web site.

Labels

We mentioned the use of labels before. With an assembler, we have the luxury of being able to create our own label names and we can use these 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 0x04               ; simple code
          movwf 0x06
```

into this...

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

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

Get a value and write it to `PORTB` to turn on a LED.

Now this is all very fine except for one thing. How does the assembler know the meaning of the labels `TurnOnLED` and `PORTB`?

The assembler has the inbuilt ability to understand all of the PIC instructions like `movlw`, and it also knows what labels are generally, but you, as the programmer, have to tell the assembler the meaning of any 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 0x04      ; value to turn on LED with RB2
PORTB           equ 0x06      ; PORTB 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 values are quite interchangeable by the way because the labels just represent simple numerical values.

For example...

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

This new piece of code is quite valid and still makes sense to the assembler, however when the PIC executes this code it will now get the literal value `0x06` and place it in `W`, and then it will get this value from `W` and place it into RAM address `0x04`. The assembler does not care what we write because its 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  "Simple Program"

          list p=16F84A              ; processor type
;
; -------------
; PROGRAM START
; -------------
;
TurnOnLed equ 0x04       ; value to turn on LED with RB2
PORTB     equ 0x06       ; PORTB address
          org 0h                    ; startup address = 0000

start     movlw TurnOnLed           ; simple code
          movwf PORTB
          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, 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 `0x04`.

When it comes across a label called `PORTB` it will substitute it for the value `0x06`. 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 0x04       ; value to turn on LED with RB2
PORTB     equ 0x06       ; PortB address
```

The assembler generates a symbol table that will look like this.

```
SYMBOL TABLE
LABEL                             VALUE
PORTB                             00000006
TurnOnLed                         00000004
```

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 an *assembler directive* and it tells the assembler to set this counter to ROM address `0h`.

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

The next code line has an address label attached to it so the assembler also adds this to its 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
PORTB                             00000006
TurnOnLed                         00000004
start                             00000000
```

Next on this code line is `movlw`. If you look in the PIC data book, the MOVLW instruction has a binary number associated with it.

Remember about computers only understanding 1' s and 0' s. This is also how the PIC understands instructions.

`MOVLW` in binary = `11 00XX kkkk kkkk`
We need to decipher this instruction 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. They can be either 1' s or 0' s and the PIC will still decode the instruction as `MOVLW`. The `kkkk kkkk` represents the 8 bits of data and will be the actual literal value.

Now knowing what the actual instruction is, the assembler will look up its symbol table and find the label called `TurnOnLED` and return with its value of 4. It will then insert this information into the `MOVLW` instruction data.

Therefore the complete instruction becomes 11 0000 0000 0100 which is 3004h.

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 16F84A 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 PORTB
```

`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 `PORTB` in it' s symbol table and returns with its value of 6.

Therefore the complete instruction becomes 00 0000 1000 0110 which is 0086h.

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 eleven bits that define the new ROM address that the processor will begin executing 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 16F84A only has a ROM address space that ranges from 0 to 1023 so a `GOTO` instruction can let the processor jump to any code memory address 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 3004h 0086h 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 cranky 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 following shows what the listing file will look like.

```
MPASM 02.30 Released           FIRST.ASM   12-31-1999  17:26:38
PAGE 1


LOC  OBJECT CODE     LINE SOURCE TEXT
     VALUE

              00001                Title  "Simple Program"
              00002
              00003                list p=16F84A   ; processor type
              00004
              00005 ;.
              00006 ;
              00007 ; -------------
              00008 ; PROGRAM START
              00009 ; -------------
              00010 ;
00000004      00011 TurnOnLed  equ 0x04 ; value to turn on LED with RB2
00000005      00012 PORTB      equ 0x06 ; PORTB RAM address
              00013
0000          00014            org 0h           ; startup address = 0000
```

```
              00015
0000 3004     00016 start      movlw TurnOnLed  ; simple code
0001 0086     00017            movwf PORTB
0002 2800     00018            goto start       ; do this loop forever
              00019
              00020
              00021            end
MPASM 02.30 Released            FIRST.ASM   12-31-1999  17:26:38
PAGE  2
Simple Program

SYMBOL TABLE
LABEL                             VALUE

PortB                             00000006
TurnOnLed                         00000004
__16F84A                          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
```

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

On the far left 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 that instruction.

This is followed by the actual assembler code.

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.

The listing file is not used by a PIC programmer to program a device, they use the hex files. The list file can be used by us to verify how the code was created. Later in your programming life you may need to use this information.

If the name of your source file was called `first.asm` then the generated list file will be `first.lst` and the generated 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 that can be viewed with any text editor program such as *Notepad*.

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

```
:0600000004308600002818
:00000001FF
```

This is the object code and has all the information that a programming device needs to write this 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.

:**06**00000004308600002818

:`06` means 6 bytes of data on the line.
Our new code was 3 WORDS long which = 6 BYTES.

:06**0000**0004308600002818

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

:060000**00**04308600002818

`00` means record type, but not on last line

:06000000**043086000028**18

`04 30 86 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

```
    3004 0086 2800
```

This is the same as our code data.

`:06000000043086000028`**`18`**

`18` is the checksum for all the data listed on this line. It is sometimes used by a programmer to make sure all the data on each line has not been corrupted when the programmer software read the data from a disk.

`:`**`00`**`0000`**`01`**`FF`

The last hex line has `00` bytes listed and `01` in the record type which means it is the last line of the file.

MPLAB

If you haven' t installed MPLAB on your computer, please do so when convenient. It can be downloaded from the **Microchip** web site, but be warned - it is big. When this program is installed you will have access to the MPLAB Integrated Development Environment by Microchip.

If you have the package we are going to do our first assembly using this program. First off, make a short cut on the desktop to the MPLAB software so that you can start it easily.

Start up **MPLAB** and click on `File - Open` and select `simple.asm` from the DIY K81 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 **16F84A** 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 a dialogue box will appear showing the compiler configurations. Just use the default values shown and press`OK` and the software will automatically run the assembler to assemble this code.

Easy isn' t it.

What ???? It created an error.

Oh my gosh, what now.

There should have been a *Build Results* window that popped up after assembly, and in it you will have this text line...

```
Error[113]   C:\DIYPROJECTS\SERIAL\SIMPLE.ASM 15 : Symbol not previously
defined (PortB)
```

What this is telling you is that `PortB` is not defined in the symbol table.

I' m guessing now that you are saying -*Well yes it is...*

Ahaaah!!! Look a little closer......

How is PortA defined??

```
PORTB     equ 0x06       ; PortB address
```

Is `PortB` the same as `PORTB`.

Close the Build Results window.

Press [`ALT F10`]. This is the same as clicking on `Project - Build Node`.

Look at the Invoke Build Tool window that opened and find the option named `Case Sensitivity`. Do you see that it is checked. To the assembler, `PortB` is *not* the same as `PORTB`.

You can do one of two things here, uncheck this item and assemble, or go back and change the offending code line. For now, leave it checked and press `OK` to start the assembly process again.

The same error line will appear. This time double click on the error line and this will open the code editor window with the cursor appearing on the offending line.

Change the line...

```
     movwf PortB
```

to...

```
     movwf PORTB
```

Now press [ALT F10] and OK to reassemble the code and this time no errors should be produced. Note that MPLAB always saves your simple.asm file before assembly takes place.

Click on Options - Development mode and make sure the MPLAM SIM - SIMULATOR function is checked, then press Reset to close the window. Click OK if a dialogue opens afterwards.

Now press [ALT - F10] again to reassemble.

Now you can 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 in a simulator window. You will notice that after this line executes...

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

...the processor jumps back to this line...

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

...and continues indefinitely.

Press the Reset Processor button to exit this mode - the button with a chip and a step on it.

Click on the editor window and put a semicolon at the start of this code line like this.

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

Now press [ALT F10] to reassemble and start stepping again.

Notice that after a few steps the processor has jumped into blank ROM space. Remember the 0x3FFF values. This is because the GOTO instruction was not assembled with the code as the assembler now thinks it is a comment. Therefore there is no instruction there to make the loop operate as intended.

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.

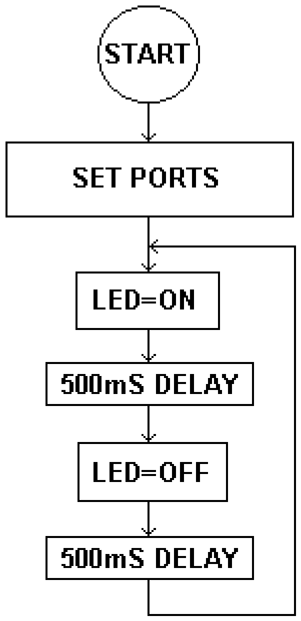
Lets now look at the code that was supplied with the kit. This code flashes the LED on PORTB pin 2 - or more simply RB2.

Close the `simple.asm` file in MPLAB and any other open windows. Now load the `flash.asm` file.

As you can see it is a bit more complicated than the `simple.asm` program.

There is a new __CONFIG directive which we will not worry about in this project and there are some new definitions and a bit more code. You should still be able to see that the general layout of the code is the same.

To understand how to formulate code for a project it is sometimes useful to create a flow chart. Here is one that describes this project.



As you can see this representation makes it a lot easier to understand what is going on.

Just follow the arrowed lines to follow the code flow.

All we are doing is setting up the port pins so they work with the circuitry that will be connected to them. Then we simply turn the LED on, wait for 1/2 a second, turn the LED back off, wait for 1/2 a second and then continue the loop forever.

We still have our `ORG` statement at the start of the code to set the ROM address where the code will start from.

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

The first thing to do is set up the ports quickly so that the pins are set up ready to control whatever circuit is connected to them.

All port pins are set as *inputs* when power is first applied to the chip.

In our project we need RB2 set as an output so that it can drive the LED connected to it. As you know there are also a lot more IO pins that are not going to be used. So what do we do with them. Well, we can' t simply ignore them because they don' t just vanish if we don' t write code for them.

If you only set RB2 as an output and leave the others set as an input you will create what are called *Floating Inputs*. This means they are trying to decide on a logic level to jump to because they are being controlled only by stray electrical charges around the circuit. This is undesirable because the internal pin circuits may be damaged and the chip may draw excess current.

You can just connect the pins to the 0V or 5V rails but if you code runs astray for some reason the pins could accidentally get changed to outputs and cause a short circuit. Imagine a pin set as a HI output and you have connected it directly to the 0V rail. - a smoking PIC.

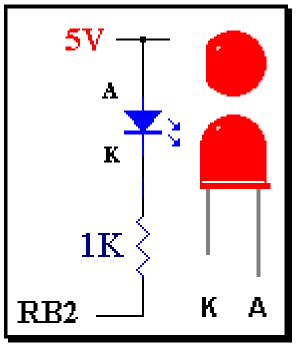
You can avoid this by using resistors to *tie* the pins to a logic level but why waste components.

Why not just set them all as outputs and set them all logic 0 or logic 1.

This is a good and simple solution but be aware that you could also short the pins out to a power source while developing the project and still cause damage.

Here is the code that sets the PORTA pins to Logic 0.

```
movlw b'00000000'      ; all PORTA pins = 0
movwf PORTA
```



]If you remember the LED circuit, the pins on PORTB must go low if the LED is on and go high if the LED is off. To make sure the LED is off initially, we can set all the PORTB pins to logic 1.

```
movlw b'11111111'      ; all PORTB pins = 1
movwf PORTB
```

The pins are reset to inputs on powerup, so we need to change them all to outputs.

Very briefly, just before we do that, to set the port pins as outputs we need to change two registers called TRISA and TRISB. These control the input/output state of all the individual pins. Each of the 8 bits in these registers corresponds to the 8 bits of the port pins. Therefore bit 0 in TRISA controls the RA0 pin, bit 7 in TRISB controls the RB7 pin etc.

Each bit that has a *logic 1* value means the corresponding port pin is an *Input*.
Each bit that has a *logic 0* value means the corresponding port pin is an *Output*.

That' s simple - 0 = Output, 1 = Input.

The PICs RAM memory is arranged in *Pages* or sometimes called *Banks*. Each page has 128 bytes of RAM and some special purpose registers. The RAM is there for us to store data in, and the other registers are there to change the way the chip operates and also to allow the code to access each of the RAM pages.

The PORT and TRIS registers are part of the RAM address space although they are some of the special purpose registers.

These can be read and written to just like a normal RAM address. Another special register is called the STATUS register. It can be used to determine the result of mathematical operations and also to tell the processor which RAM bank is active. The bit in the STATUS register that does this is called the RP0 bit which is actually bit 5.

When the RP0 bit is logic 0, the processor can access RAM page 0.
When the RP0 bit is logic 1, the processor can access RAM page 1.
PORTA and PORTB are addressed in RAM Page 0.
TRISA and TRISB are addressed in RAM Page 1.

Therefore, if we want to modify the TRIS registers to set the PORT pins as outputs we first must make sure the RP0 bit = 1.

This next instruction does that.

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

Now we can change the TRIS registers to make all pins outputs.

```
          movlw b’00000000’     ; all PortA = outputs
          movwf TRISA
          movlw b’00000000’     ; all PortA = outputs
          movwf TRISB
```

Notice the binary values and how they make it easy to see what pins are inputs and outputs.

Now that we have set the TRIS registers we can return to RAM Page 0.

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

Please don’ t get confused by things like this. Look through the 16F84A data sheet and verify that the addresses that we assigned to the registers etc. are what is written there. Also check out what STATUS and other registers do. Pretty soon you will become familiar with the way it works.

Now, the next part of the flowchart shows that we turn the LED on. Therefore we set RB2 to logic 0 and leave the others set to logic 1. If you remember the HEX numbers, that just means write 0xFB to PORTB and we can do it in binary like this to see that we are setting RB2 to 0.

```
MainLoop   movlw b'11111011'         ; LED on RB2 = on
           movwf PORTB
```

This is also the start of our code loop which is why the MainLoop label is there.

The next thing we do is create a 1/2 second delay.

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 this project runs with a clock speed of 4MHz. That is 4 million *clock 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 `0x00`.

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

Inside the 16F84A, there are a lot of 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`. As you now know, we can define the label like this so that we can use it in our program.

```
DelayL     equ 0x0C     ; delay register LOW byte
```

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

The `DECFSZ DelayL,f` instruction means to decrement the contents of `DelayL` by 1 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.

It works like this...

Sequence of events for `DECFSZ DelayL,f`

`W` Register = `0xA0`
`DelayL` = `0x00`
Value is read into the Arithmetic Logic Unit (ALU) = `0x00`
Value in ALU is then decremented by 1 = `0xFF`
`DelayL` = `0xFF`
`W` Register = `0xA0`

Sequence of events for `DECFSZ DelayL,w`

`W` Register = `0xA0`
`DelayL` = `0x00`
Value is read into the Arithmetic Logic Unit (ALU) = `0x00`
Value in ALU is then decremented by 1 = `0xFF`
`DelayL` = `0x00`
`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).
```

`f` and `w` are labels as well, but the assembler knows what they are.

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 `0x0C`.

```
      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 `0x00`. Then it will be decremented by one and it will have a value of `0xFF`. This new value is not equal to `0x00`, 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 `0x00` 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 `0x00`. It will then take two instruction cycles. A `GOTO` instruction always takes two instruction cycles to complete.

If *any* instruction changes the value of the *Program Counter* then 2 instruction cycles will be used.

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.

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*.

Nested loops 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.

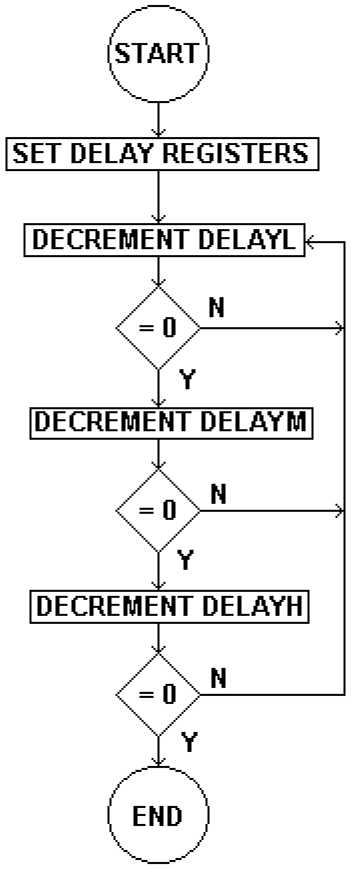
Let’ s define these registers so we can use them.

```
DelayL    equ 0x0C       ; delay register LOW byte
DelayM    equ 0x0D       ; delay register MID byte
DelayH    equ 0x0E       ; delay register HIGH byte
```

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

```
          clrf DelayL              ; clear DelayL to 0
          clrf 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 `0x00` 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 `0x00` and then the second `goto WaitHere` instruction is skipped. `DelayH` is then decremented and it will equal `0x02`.

The third `goto WaitHere` will execute because `DelayH` does not equal `0x00` yet. This triple loop will continue until `DelayH` equals `0x00` 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.

If you remember the original flow chart, we have to turn the LED on, wait, turn the LED off, wait etc etc.

Notice that there are two time we need to wait. In those instances, we need to insert all that delay code above. Imagine if we had to have 100 delays.

That would soon eat up all our available code space. An easier way to do the delay, is to create what is called a *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.

This is how we can turn the delay code into a subroutine.

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

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. 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.
Lets now complete the first section of the loop...

```
MainLoop   movlw b'11111011'         ; LED on RB2 = on
           movwf PORTB
           call Delay500            ; execute a 500mS delay
```

Can you figure out the next section which is to turn off the LED and wait. I' m sure you can.

Perhaps you could write your idea down on this page before you move on.

No peeking....

Just make RB2 = logic 1.

```
MainLoop   movlw b'11111111'         ; LED on RB2 = off
           movwf PORTB
           call Delay500            ; execute a 500mS delay
```

Now we just tell the processor to jump back to the start of the code loop which will make it continue forever.

```
          goto MainLoop             ; do this loop forever
```

Immediately after this instruction you can place the `Delay500` subroutine code and the program is complete.

If you now look at the `flash.asm` code listing in MPLAB you should be able to see how it works.

To simulate it just type [`ALT F10`] and `OK`.

Now step through the code. You can also single step the code faster by pressing `F7`.

I' ll bet you get bored waiting for the `Delay500` subroutine to complete. MPLAB will take forever to run this code section because it is so much slower than the real PIC. We know the delay subroutine works, so to speed things up so we can check the code flow, change this line...

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

...to this...

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

This has the affect of returning from the subroutine as soon as it is called. The assembler will ignore everything after the first semicolon.

To simulate it again, just type [`ALT F10`] and `OK`, and start stepping.

Open up the `StopWatch` by clicking on `Windows -> StopWatch`.

You will be able to see how long each instruction takes at a 4MHz clock speed.

When you are satisfied that the code operates as normal change the `Delay500` code line back to the original state.

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

Assemble it again using [`ALT F10`].

Now run **diyk81.exe**. and press `Program` and select the `flash.hex` code to program into the chip. Now press `Run`. The LED should be flashing as it did the first time.

Use the blank section of this page and write down how you think the code should be changed to make *all* the LEDs flash.

Remember that the LEDs are connected to `PORTB` pins `RB2`, `RB3`, `RB4`, `RB5`, and `RB6`.

See over the page if you are stuck.

```
MainLoop   movlw b'10000011'
           movwf PORTB
           call Delay500            ; execute a 500mS delay

           movlw b'11111111'
           movwf PORTB
           call Delay500            ; execute a 500mS delay
```

Pretty simple isn’ t it.

Modify the `flash.asm` code then try it out and see.

How can you convert this code into a single instruction??

```
           movlw b'00000000'
           movwf PORTB
```

Check out the data sheet to find out and look in the instructions section.

(Hint: the instruction is `cl.. PORTB`)

If you are stuck, turn the page....

```
        clrf PORTB
```

For some additional exercises.....

Try to make the LEDs count up in binary.
Try to make LEDs count down in binary.
Try to makes the LEDs come on one at a time moving from left to right.

There are code examples for these exercises in the installation directory for you to look at. Try to do them yourself and use the examples if you get confused.

```
binaryup.asm

binarydn.asm

binarylr.asm
```

Two of these programs use the `XORLW` instruction to make sure the data written to `PORTB` is correct. Try to figure out why it was used.

```
XOR TRUTH TABLE

    A     B    OUT

    0     0     0
    0     1     1
    1     0     1
    1     1     0
```

Hint: Notice how `A XOR 1 = opposite A`

If `XORLW` wasn' t used how would you increment or decrement the `Counter` value to make the program work. As you will find out, it' s much better to use the XOR instruction.

CONFIG

The `__CONFIG` statement at the start of the code is used to control the *Configuration Fuse* register. This is a special location that controls how the chip operates. Things like *Oscillator Type, Watch Dog Timer, Programming Mode* and other features. These are listed in the data sheets. The value used...

```
   __CONFIG 0x3FFB
```

... is a specially selected value for this project. It selects *RC Oscillator, Watch Dog Disabled* and no *Code Protection*.

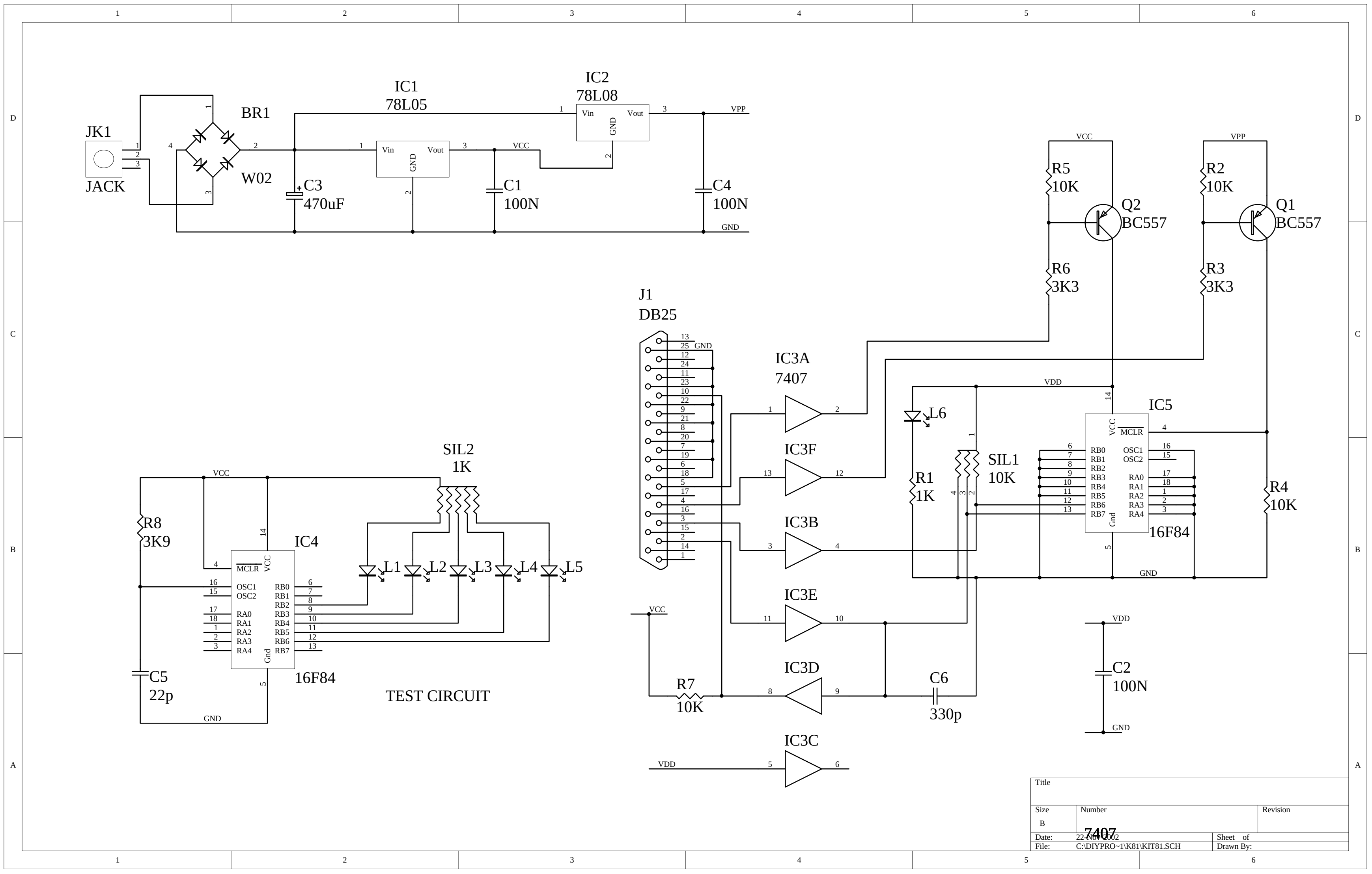
RC mode was selected because the PIC clock source connected to the demonstration socket is a simple **R**esistor/**C**apacitor circuit.

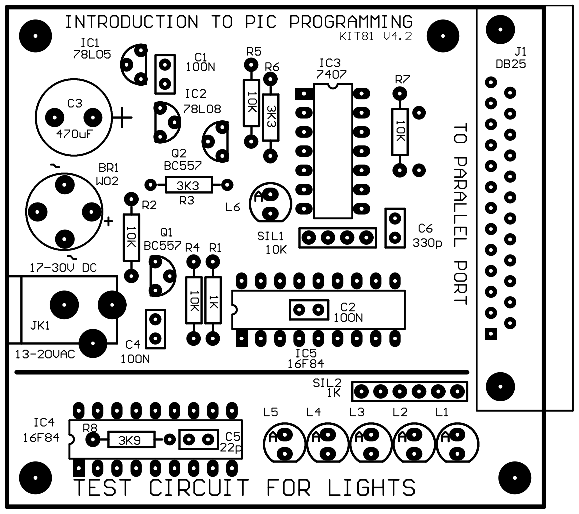
Please do not alter this value as the PIC may stop operating when you run the code. If you accidently changed the value, change it back and reassemble. After programming again, the chip should run normally.

Cheers and happy programming.

PARTS LIST

Used	Part Type	Designators	Description
SEMICONDUCTORS			
1	PIC16F84A	IC4	PIC Processor
1	74LS07	IC3	HEX OC buffer
1	78L05	IC1	Regulator
1	78L08	IC2	Regulator
2	BC557	Q1, Q2	PNP Transistor
1	W02	B1	Bridge Rectifier
1	RB2	L2	5mm LED
1	RB3	L3	5mm LED
1	RB4	L4	5mm LED
1	RB5	L5	5mm LED
1	RB6	L6	5mm LED
1	Program/Verify	L1	5mm LED
RESISTORS all 1/4W 5%			
1	SIP 10K	SIL1	
1	SIP 1K	SIL2	
1	1K	R1	
2	3K3	R3, R6	
1	3K9	R8	
4	10K	R2, R4, R5, R7	
CAPACITORS			
1	330p	C6	Ceramic
1	22p	C5	Ceramic
3	100N	C1, C2, C4	Ceramic
1	470uF	C3	Electrolytic 35V
MISCELLANEOUS			
1	BLANK PCB		PCB1
2	IC SOCKET		18 pin
1	IC SOCKET		14 pin
1	PARALLEL	CN1	Male RA DB25
1	JACK	JK1	3 Pin Power Jack


JK1
JACK
BR1
W02
IC1
78L05
IC2
78L08
Vin
Vout
GND
VCC
VPP
C3
470uF
C1
100N
C4
100N
GND
R5
10K
Q2
BC557
R2
10K
Q1
BC557
R6
3K3
R3
3K3
J1
DB25
IC3A
7407
IC3F
IC3B
IC3E
IC3D
IC3C
VDD
L6
R1
1K
SIL1
10K
IC5
16F84
MCLR
OSC1
OSC2
RB0
RB1
RB2
RB3
RB4
RB5
RB6
RB7
RA0
RA1
RA2
RA3
RA4
Gnd
R4
10K
R7
10K
C6
330p
C2
100N
SIL2
1K
R8
3K9
IC4
16F84
L1
L2
L3
L4
L5
C5
22p
TEST CIRCUIT
Title
Size
B
Number
7407
Revision
Date:
22-Nov-2002
Sheet of
File:
C:\DIYPRO~1\K81\KIT81.SCH
Drawn By:


INTRODUCTION TO PIC PROGRAMMING
KIT81 V4.2
IC1 78L05
C1 100N
IC2 78L08
C3 470uF
R5 R6 10K 3K3
IC3 7407
R7 10K
J1 DB25
TO PARALLEL PORT
BR1 W02
Q2 BC557
3K3 R3
L6
R2 10K
Q1 BC557
R4 R1 10K 1K
SIL1 10K
C6 330p
17-30V DC
JK1
13-20VAC
C4 100N
C2 100N
IC5 16F84
SIL2 1K
L5 L4 L3 L2 L1
IC4 16F84
R8 3K9
C5 22p
TEST CIRCUIT FOR LIGHTS