PIC Tutorials

(EASYPIC4 version)

INTRODUCTION.

These tutorials have been developed by John Becker and published in the Everyday Practical Electronics magazine as a series of articles in March to June1998. The present version has been modified to be suitable for classroom instruction. The original tutorials were based on the TASM assembler. The programs were tested on an evaluation board which consisted of the PIC 16F84 and some switches for inputs and LEDs to display the output ports. The present version uses the 16F877A microcontroller. The TASM has been replaced by the MPASM assembler, widely available as a part of the MPLAB Integrated Development Environment (IDE) programming environment. The MPLAB IDE, amongst others, also allows software simulation to be performed without using any microcontroller hardware. However, it has to be realised that the software simulation under the MPLAB is not entirely realistic when compared with the hardware run. The main differences being in terms of speed and interfacing to external devices. For this reason these tutorial continue to be based on an evaluation board, although the one used here, the EasiPIC board from Mikroelektronika, is different to the original one. This meant that some of the tutorial had to be modified.

WHAT'S A PIC?

A PIC chip, in this context, is a microcontroller integrated circuit manufactured by Arizona Microchip. A microcontroller is similar to a microprocessor but it additionally contains its own program command code memory, data storage memory, bi-directional (input/output) ports and a clock oscillator. Many microprocessors require the use of additional chips to provide these requirements; microcontrollers are totally self-contained.

The great advantage of microcontrollers is that they can be programmed to perform many functions for which many other chips would normally be required. This not only makes for simplicity in electronic designs, but also allows some functions to be performed which could not be done using normal digital logic chips - i.e. circuits for which, previously, a microprocessor would have been required.

There are many families of PIC microcontroller available, ranging from those which can only bc programmed once, to those that can be repeatedly reprogrammed. There are two basic families of the latter: those that require an ultra-violet light unit to erase their previous data before being reprogrammed, and one device family which is electrically erasable.

In the latter PIC microcontroller family is the P1C16F877A, which is the device which we shall use throughout this series of articles. It has been chosen because of its ease of reprogramming. It is an EEPROM (electrically erasable programmable read only memory) device. This means that it can be rapidly reprogrammed as often as you wish, without the need for ultra-violet erasing.

Much of the information about the commands which we present here is, in most instances, applicable to other members of the PIC family. Once you understand a P1C16F877A you should have no difficulty applying your knowledge to other PICs.

EASYPIC4 Demonstration Board

The demonstration board provided for these tutorials enables one to try out the various features of most of the 8 bit PIC microcontrollers. In order to use the board with the 16F877A microcontroller, the 16F877A chip should be inserted into the socket marked DIP40. Only one microcontroller chip can be used at any time, so it is important to ensure that no other sockets are occupied. The board connection of the two OSC1 pins on the DIP40 socket is to the oscillator socket marked OSC1. This should have either an HS oscillator crystal or, if so desired, can be used as an input being driven from a standard TTL voltage level external oscillator. In the latter case the crystal needs to be removed. There is no provision on the board to use the internal RC oscillator mode. The 16F877A has five ports known as PORTA, PORTB, PORTC, PORTD and PORTE. The pins of these five microcontroller ports are connected to the correspondingly marked LEDs and also to the press switches. The press switches can be used as external inputs and can be configured to provide either "1" or "0" when pressed. The appropriate configuration is performed by the jumper JP17. The LEDs can be used for visible monitoring of the output ports. They can be disconnected, if so required, from the port by means of the slide switches SW2. The MCLR pin is used to program the microcontroller in-situ. All pins, except the MCLR, are brought out to the external ICD connectors at the edge of the board.
The power to the board can be provided from the USB connection to the PC. This requires the Supply Select jumper (the closest to the Power Supply connector) to be configured for USB power connection.
The detail schematic showing all connections can be found in the User Manual .

Software

The software used to develop the programs for the 16F877A microcontroller is the MPLAB IDE. This provides a platform for developing PIC microcontroller applications in a low level assembler language, or high level languages such as C and Basic. The present tutorial is based on the Microchip assembler called MPASM. Our program development process will consist of the following stages: Firstly, a new Project is created. This stores the required software/hardware environment, such as the language suite used, the type and the configuration of the microcontroller etc. Next, the assembly code is entered and saved in a form of a source file (with extension .asm). The file is added to the Project. (In more complex project, other files, including libraries and linker files may be required to be added to the Project) The next stage is called the Build. This is when the source file(s) is converted

by the MPASM assembler into a programmable hex file (extension .hex). This file is in a form that can be used by an IC Programmer device to download the machine code into the microcontroller's programme memory.
If the Project environment is configured to run in the Debug Mode, the Build also produces a simulation file that can be used by the MPLAB's software simulator to verify the correct operation and to debug any faults
The programming of the microcontroller can be done by either placing the chip into a IC socket of separate proprietary Device Programmer or with the IC in-situ. We shall perform the programming in-situ via the the EASYPIC4 board USB connector. To do so we shall use the proprietary Mikrolektronika software called EasyFlash.

Testing the EASYPIC4 Board.

To test the operation of the board we shall run a little hex file named buttons.hex. This has been already pre-assmbled from the the source code that is given at the end of this section. It demonstrates some functionality of the board. It treats the press switches RB0 to RB4 as microcontroller inputs and the LEDs RD0 to RD4 as outputs. The test program is very simple. The 16F877A keeps reading the five input pins and passes the values straight to the output. Thus, when one presses a press switch, say RB2, the corresponding LED (RD2) should light up.

To download the hex file into the chip, ensure that the EASYPIC4 board is connected to the PC via the USB cable. Start the EasyFlash program. Select the Load button and navigate to the directory where the buttons.hex file has been saved. Once the file has been loaded it can be downloaded to the 16F877A by selecting the Write button.

Before testing the board, disconnect the PORTB LEDs by the top leaver on the LED SW2 slide switch and power up the external oscillator to provide the CPU clock. Set the frequency to about 1k Hz. Observe what happens as you reduce the external frequency to few Hz. and try to explain it.

Source code for the buttons.asm file:

```
; This Code may be used to test the EASIPIC4 board
; Operation: PORTB: input connected to Buttons RB0 to RB4,
;            PORTD: output connected to LEDs RD0 to RD4
;            Lights up the corresponding PORTD LED depending on
;            the button pressed.

list  P = PIC16F877A R = DEC
include P16F877A.inc
    __config(_CP_OFF & _PWRTE_OFF & _XT_OSC & _WDT_OFF )  ; note,  two underscores
    errorlevel -302                                       ; ignore "error" when storing to Bank1

#DEFINE PAGE0 bcf STATUS,5
#DEFINE PAGE1 bsf STATUS,5

        org h'0005'
        clrf    PORTB
        clrf    PORTD
        PAGE1
        movlw 0xff
        movwf  TRISB
         clrf    TRISD
        PAGE0
         clrf    PORTD

LOOPIT:
       movf    PORTB, W
       movwf  PORTD
       goto    LOOPIT
        end
```

TUTORIAL I

CONCEPTS EXAMINED

Minimum software requirement
Port default values
Instruction ORG
Instruction END
Instruction .AVSYM

The absolute bare minimum requirements for any PIC 16F877A program which is to be compiled under MPASM are shown in Listing I. In fact, none of the statements in this listing have anything directly to do with a functioning software program. Two are aimed directly at the MPASM assembler; the others are comments to the programmer, or other reader.

LI5TING I - PROGRAM TUTI

```
; TUT1.ASM
; minimum requirement software
    list      p=16f877a  ; list directive to define processor
    #include  <p16f877a.inc>  ; processor specific variable definitions
    __CONFIG _CP_OFF & _WDT_OFF & _BODEN_OFF & _PWRTE_ON & _XT_OSC & _WRT_OFF & _LVP_ON
    __CONFIG  CPD_OFF
    ORG H' 0005'  ; Start of program memory
          ; (your program goes in here)

     END  ;     final statement
```

Comments must always be preceded by a colon (;) so that MPASM does not try to treat them as program commands (machine language instructions for the PIC).
Comments may appear anywhere within the program, and in any position where they do not interfere with a program command.
It is convenient, though, to place them tabulated a short distance beyond the end of program command lines.
To take Listing 1 in detail, you will see that it starts with two comments, identifying the listing and its function:

; TUT1.ASM
; minimum requirement

Next come commands which are aimed at the MPASM assembler. They are called assembler directives. The **list** directive specifies the processor. The **include** directive inserts the file p6f877a.inc that contains various symbol definitions that are specific to the 16F877A. The **__CONFIG** directive (note two leading underscores) defines the so called configuration bits that determine the mode of operation. All the above directives have to be present in in the heading of every assembler source file so need to be repeated it parrot-fashion in any software you write will normally suffice unless interrupts are involved:

The directive **ORG** specifies the program memory address (position) within the PIC at which a particular set of subsequent commands is to be placed. The command consists of the **ORG** (Set Program Origin) statement followed by a number, which may be expressed in hexadecimal as shown here . Note that MPASM accepts commands in either capitals or small lettering, i.e the commands are not case sensitive. The bold formatting is only to emphasise the commands within the current document and is not used in the actual source file. The symbol H stands for the hexadecimal number that is enclosed within a pair of single quotation marks.

When the PIC is put into programming mode, it sets its internal Program Counter to 0000, which is the reset vector address. The - Program Counter (PC) is an internal PIC register that holds the memory address of the next instruction to be executed. It is a 13 bit wide so it can address up to 2^{13}(i.e. 8129) different program memory locations. Addresses 0001 to 0003 are normally not of any use to the user. Address 0004 is the interrupt vector which is set by the first line in the assembly code. The interrupt vector address at 0004, and the subject of *interrupts* in general, will be dealt with in Tutorial 28. Ignore the concept for the moment. The user program starts normally at 0005.

The next line in the listing is a comment and is aimed at you, the reader: it tells you where your program is to be written. This will become evident as we progress through the example listings.
The final statement (**END**) is another directive which tells the MPASM assembler that it has reached the end of the source file. It MUST always be placed at the very end of any listing which is to be passed through MPASM.

INCAPABLE?

You might think that when TUT1. ASM is compiled by MPASM and loaded into the PIC as TUT1 .hex, that the PIC will be incapable of doing anything - it hasn't been told of anything to do. Almost true, but not quite! The true part of the statement is that our listing does not tell the PIC what to do. The directives ORG and END instruct the MPASM assembler, that is the software that translates our code into the binary form that can be understood by the PIC. They define the first and the last memory location where the program goes. They are thus not true CPU (machine code) instructions but assembly directives. There are no CPU instructions in the above source code and thus no machine code is entered into memory location defined by the ORG directive.

Although the PIC has been not given anything to do by us it has been told in manufacture to adopt certain "default" conditions when first switched on . One of these conditions is that Port B and Port D are configured (set) to act as inputs. Why the inputs rather than outputs ? Think of it.

EXERCISE I

1.1. After starting the MPLAB open a new Project with a name of your choice. A Project is just an administrative file that keeps all the information on the whereabouts of various files and the chip settings that you use. The easiest way to define a new Project is to use the Project Wizard from the Project Menu. When prompted by the Wizard, from the device list select the PIC 16F877A device. Next, the Language Suite should state that you are going to work with the Microchip MPASM Language Suite. When any of the three components of the Suite (i.e. mpasmwin.exe, mplink.exe or mplib.exe) is highlighted in the Toolsuite Contents window its location (path) should appear in the Location window beneath. If this is not the case you will have to use the Browse button to point the MPLAB to location of the program component. Next, you will choose the name and the location of your Project files. The Project file that you choose is saved with the extension .mcp. As we have not created our source file there is nothing to add, so we can skip the next step and then exit the Wizard.

1.2. Next we need to specify the 16F877A configuration that will be saved into the chip when the program is downloaded to it. From the Configure menu select Configuration bits. Unclick the top box as we do not wish to specify the configuration as in our source code. Select the oscillator. For crystal oscillator select XT (XT cut). The same setting is also used when the chip is driven by an external oscillator. Make sure that all the remaining settings are turned off. Close the Configuration Bits window.

1.3. Now let us create a new source file. From File -> New start a new file that will hold your source code. Copy the Listing 1 and save it under the same name which as the project name given earlier. In order to indicate to MPLAB that this is an assembler source file you need to save it with the extension .asm. At this stage the source file you have just saved is not yet being recognised by MPLAB as a part of your project. To do so you will need to tell the MPLAB to add this file to the existing project. From the Project menu select Add New File to Project.. and proceed to add the new source file just saved.

1.4. Before we build we can choose whether to produce a minimum length efficient code (so called Release) or have a much more larger code required for software simulation and debugging. From the Project menu select Configuration Options and Debug..

1.5. Now we are ready to assemble the source file. From the Project menu select Build All. If there are any errors in your source code the MPASM will inform you and you will need to correct them all before you can proceed.

1.6. MPLAB allows you to simulate the assembled code, before downloading the program hex file to the device The binary machine code can be viewed in the simulator MPLAB SIM. To do so, go to the Debugger menu and Select Tool MPLAB SIM. Additional items appear on the Debugger menu (and also on the Tool bar) that will allow you to run the whole of the program or to debug it by stepping in, stepping out etc. in your source code.

1.7. Let us see what is currently stored in the Program Memory. From the View menu open the Program Memory window. It shows the content of the memory locations starting from the memory address 0000. The listing 1 does not contain any machine instructions so you will see all the memory locations holding the same value of 3FFF. This represents a binary number 11 1111 1111 1111. No coincidence here, this is the largest number each program memory address can hold. These fourteen bits represent the width of the PIC 16F877Aprogram memory. Other PICs have different memory sizes.

1.8. The right hand column of the Program Memory display represents the MPASM mnemonic for the machine code that is held there. You will see the MPASM instructions **ADDLW 0xff** in every memory location. This is a machine instruction which tells the PIC to Add the Literal (e.i.. a constant) 0xff into one of the internal registers, the so called Working register denoted W. We shall meet this instruction later on. The question is why is it in the program memory even though it does not appear in our source code? Here is the explanation. When the power is applied to the PIC all program memory bits are set (high), hence the value 3FFF. This value also represents the machine instruction **ADDLW 0xff.** So what happens is this. The PIC starts executing instructions from address 000. The first instruction adds hexadecimal 0xff to the working register. The PC increments to point to memory address 001. There it finds the same instruction, so further 0xff is added to the working register and so on and on, until program counter points at last physically implemented memory location 8192. What happens next ? Remember, the PC can count up to 8192 which is a hexadecimal 1FFF. Thus when the PC advances to 0x1FFF, which happens to represent the binary 1 1111 1111, the PC counter overflows and starts addressing memory locations from the zero address. Thus even though our listing 1 does not contain a single machine language instruction, the PIC will be adding 0xff to its internal register until we switch it off.

1.9. We can simulate this in the MPLAB. With the program memory displayed select the command run from the Debug menu. Take the option of a single step (F7). Each press of the F7 key will advance the PC to execute the instruction in the next memory location. This can be seen by the cursor moving downwards.

EXERCISE II

1.1. Now we can download the hex file to the device. Make sure the EASYPIC4 board is connected by the USB cable to the PC. Run the PIC Flash programme. Press the Load button and navigate to the project directory where you have saved your hex file. When the file is read the PICFlash software will try to determine the Bit configuration from the hex file. Check that the correct configuration appears in the relevant windows. Thus we should have 16F877A in the Device window. The configuration bits should show: Oscillator: XT, Watchdog Timer: OFF, Power Up Timer: OFF, Code Protect: OFF. If not, correct it now. To download the hex file into the PIC press the Write button. The program now resides in the chip.

1.2. Now, we are ready to test the chip. Ensure that the external oscillator is connected to the OSC1 socket or an equivalent connection is made on the board. Do you see any LED lighting up? No, you should be able to explain why. Is then nothing happening ? Well, you know that the PIC is busy adding 0xff, but you cannot see it in real life.

1.3. Let us try the simulator. Back in the MPLAB, open the Program Memory window. In the Debugger menu Reset the program. The green arrow in the Memory Window shows the location of the current instruction being executed. It should point at the memory location zero. Now, in the Debugger menu press repeatedly the Step Over (F8) button and you should see the arrow advance as each new instruction is being executed.