EPE PIC TUTORIAL V2

JOHN BECKER   PART ONE

Quite simply the easiest low-cost way to learn about using
PIC
Microcontrollers!

WHAT IT'S ALL ABOUT

AT the time that the original *EPE PIC Tutorial* was published in March to May 1998, letters and phone calls to *EPE* had been showing that interest in Microchip's PIC microcontrollers had become intense. Many readers were asking for more information on how to use these devices in designs of their own invention.

In the words of one reader, "I find the PIC data sheets too skimping on everyday detail, and the published software too complex. Please show me how to get to grips with the *essence* of PICs. Tell me, step-by-step, how to get started with writing simple programs, how to just turn on a single light emitting diode, for example. Then take me forward from there."

It was to meet this demand that the original *EPE PIC Tutorial* was published. Its success resulted in a CD-ROM version being produced commercially as *PICtutor*, complete with its own ready-built development board. Recently that version was upgraded to become *Assembly for PICmicro V2* along with its Version 2 PICmicro MCU board (see elsewhere in *EPE* for details). Many thousands of people have learned to program PICs through these several versions.

However, we frequently receive requests from readers for PIC programming education that is available at minimal cost. The aim of this three-part *EPE PIC Tutorial V2* series is to do just that, as detailed more fully on the next two pages. In a nutshell, though, it gets you inexpensively started with PICs, and does so "step-by-step".

This CD-ROM requires Adobe Acrobat® Reader™ Acrobat Reader v5.05 is included on the CD-ROM.

PIC RESOURCE

© Wimborne Publishing Ltd 2003
www.epemag.wimborne.co.uk

EPE PIC TUTORIAL V2
01202 873872
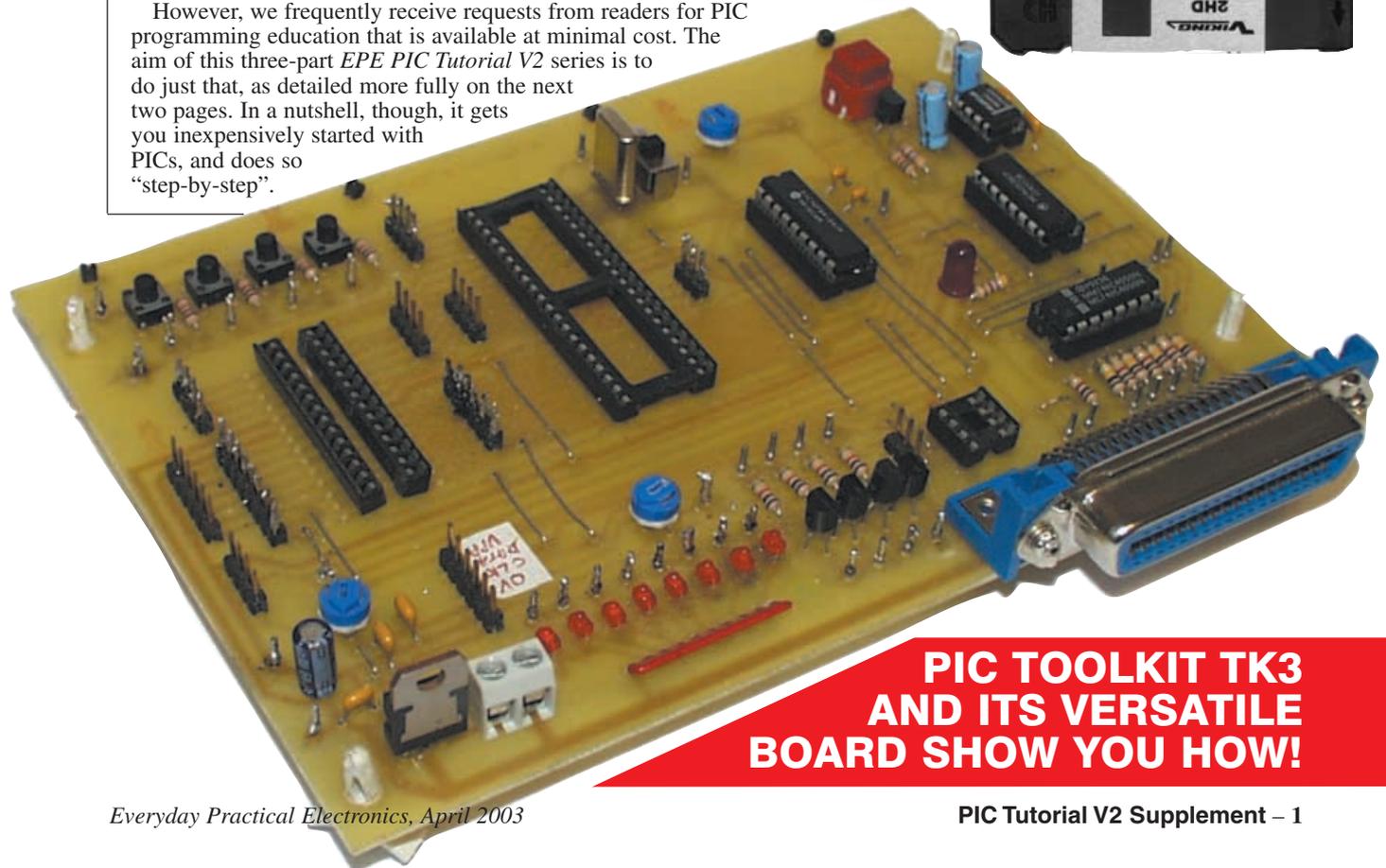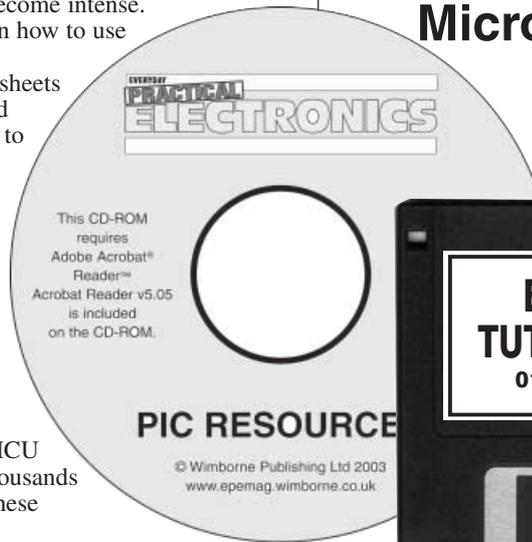
PIC TOOLKIT TK3 AND ITS VERSATILE BOARD SHOW YOU HOW!

THIS REVISION

Over five years on from the publication of the original *EPE PIC Tutorial*, a number of things have changed, yet at the same time the basics of programming PIC microcontrollers have not.

This revision is thus a mixture of the old and the new. The old aspect is that the commands used to program PICs remain the same. The new aspects, though, are several:

The original *EPE PIC Tutorial* illustrated its example programs in a programming dialect known as TASM. This dialect is usable with a variety of tables whose contents can be modified to suit many types of microcontroller and microprocessor. It had been modified to suit PICs by reader Darren Crome.

This revision now has its programming examples written in Microchip's own PIC programming dialect, MPASM. This dialect is the "industry standard" and thus has far wider appeal than TASM, although the basic differences between the two are slight.

Secondly, the original *EPE PIC Tutorial* concentrated on the now-obsolete PIC16C84 as being the target device. This microcontroller effectively became replaced in 1997 by the pin-for-pin compatible PIC16F84, which is an equally excellent device to use to illustrate PIC programming techniques. More recently the PIC16F84A has arrived on the scene. The PIC16F84 and PIC16F84A (two of the devices in the PIC16F8x family) can be used interchangeably in this *EPE PIC Tutorial V2* (referred to from now on as the *Tutorial*).

FAMILY MATTERS

Once you know how to program a PIC16F84 you are well equipped to write programs for other PICs, although there are some minor differences in the way that the various PIC families handle some of the same functions.

Apart from the PIC16F8x family, two other PIC families are eminently suited to hobbyist constructors, notably the PIC16F87x and PIC16F62x families (although they are not immediately suited to this *Tutorial*). However, in the final part of this three part series, basic differences between the way that the PIC16F8x, PIC16F87x and PIC16F62x families do the same thing are highlighted and the Tutorial programs can be readily modified to run on these devices. Examples of some useful routines specific to the PIC16F87x family are included.

It is stressed, though that this *Tutorial* does not attempt to be a full tutorial on every aspect of the three families. Nor does it examine specific aspects of some other PIC families whose functions are more advanced than most readers probably require.

Also, the *Tutorial* does not teach the use of Microchip's MPASM and MPLAB programming software, and it does not cover any PIC variant or dialect that is programmed in versions of BASIC.

An important aspect of this revision is that it has been designed for use with the *EPE PIC Toolkit TK3* printed circuit board and software (published Oct/Nov 2001), both of which, plus their two texts, you need in order to get full benefit from this *Tutorial*. See the Resources panel for details of obtaining them. Note that these

are included on the *EPE PIC Resources CD-ROM*. From hereon the software and p.c.b. are jointly referred to simply as *TK3*.

It should be noted that *TK3* was written to run under Windows 95, 98 and ME. To run it under Windows NT, XP and 2000 the software must be used as described in Mark Jones' article *Using TK3 with Windows XP and 2000* of Oct '02. This article is also carried on the *EPE PIC Resources CD-ROM*.

In keeping with the original *Tutorial*, we assume in this series that you have no previous knowledge of PICs and their programming. We thus start as we did before, by explaining the basic nature of a PIC microcontroller.

WHAT IS A PIC?

A PIC chip, in this context, is a microcontroller integrated circuit manufactured
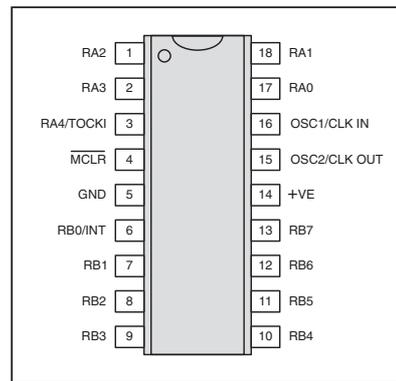
Fig.1. Pinouts for the PIC16F8x family.

by Microchip. When asked about the name's origin, Microchip's Technical Department

BASIC PIC16F84 SPECIFICATIONS

35 single-word commands (see Table 1)

1K × 14-bit EEPROM program memory

68 × 8-bit general purpose SRAM registers

15 special function hardware registers (see Table 2 later)

64 × 8-bit EEPROM data memory

1000 program memory erase/write cycles (typical)

10,000,000 data memory erase/write cycles (typical)

Data retention >40 years

5 data input/output pins, Port A

8 data input/output pins, Port B

25mA current sink maximum per pin

20mA current source maximum per pin

80mA maximum current sunk by Port A

50mA maximum current sourced by Port A

150mA maximum current sunk by Port B

100mA maximum current sourced by Port B

Total power dissipation 800mW

8-bit timer/counter with 8-bit prescaler

Power-on reset (POR)

Power-up timer (PWRT)

Oscillator start-up timer (OST)

Watchdog timer (WDT) with own on-chip RC oscillator

Power saving Sleep function

Code protection

Serial in-system programming

Selectable oscillator options:

RC: low cost RC oscillator

XT: standard crystal/resonator (100kHz to 4MHz)

HS: high speed crystal/resonator (4MHz to 10MHz) (to 20MHz for 'F84A)

LP: power-saving low frequency crystal (32kHz to 200kHz)

Interrupts:

External, RB0/INT pin

TMR0 timer overflow

Port B RB4 to RB7 interrupt on change

Data EEPROM write complete

Operating voltage range: 2·0V to 6·0V (to 5·5V for 'F84A)

Power consumption:

<2mA @ 5V, 4MHz

15μA typical @ 2V, 32kHz

<1μA typical standby @ 2V

TUTORIAL CONCEPTS EXAMINED

TUTORIAL 1:
Minimum commands needed
Port default values
Instruction ORG
Instruction END
Command GOTO
Program TK3TUT1.ASM

TUTORIAL 2:
Clock cycles
File registers
Bits
Bytes
Set
Clear
Command CLRF
Command CLRW
Command BSF
Command BCF
Ports and Port directions
Register STATUS
STATUS register bit 5
Banks 0 and 1
Program TK3TUT2.ASM

TUTORIAL 3:
Names in place of numbers
Case sensitivity
Labels
Repetitive loop
Instruction EQU
Program TK3TUT3.ASM

TUTORIAL 4:
Command MOVLW
Command MOVWF
Command RLF
Command RRF
Command BTFSS
Command BTFSC
Instruction #DEFINE
Instruction BANK0
Instruction BANK1
Register PORTA
Register TRISA
Register PORTB
Register TRISB
Register PCL
Naming numbers
Bit naming
Program counter
STATUS register bit 0
Carry flag
Bit codes C, F, W
Bit testing
Conditional loop
Pin protection
Program TK3TUT4.ASM
Program TK3TUT5.ASM

TUTORIAL 5:
STATUS bit 2
Zero flag
Bit code Z
Command MOVF
Program TK3TUT6.ASM

TUTORIAL 6:
Command INCF
Command DECF
Command INCFSZ
Command DECFSZ
Counting upwards
(incrementing)
Counting downwards
(decrementing)
Use of a file as a counter
Program TK3TUT7.ASM

TUTORIAL 7:
Switch monitoring
Command ANDLW
Command ANDWF
Command ADDWF
Command ADDLW
Nibbles
STATUS bit 1
Digit Carry flag
Bit code DC
Program TK3TUT8.ASM

TUTORIAL 8:
Increasing speed of
TK3TUT8
Bit testing for switch status
Program TK3TUT9.ASM

TUTORIAL 9:
Responding to a switch press
only at the moment of
pressing
Program TK3TUT10.ASM

TUTORIAL 10:
Performing functions
dependent upon which switch
is pressed
Use of a common routine
serving two functions
Program TK3TUT11.ASM

TUTORIAL 11:
Reflecting PORTA's switches
on PORTB's l.e.d.s
Command COMF
Command SWAPF
Inverting a byte's bit logic
Swapping a byte's nibbles
Program TK3TUT12.ASM
Program TK3TUT13.ASM
Program TK3TUT14.ASM

TUTORIAL 12:
Generating an output
frequency in response to a
switch press
The use of two port bits set
to different input/output
modes
Command NOP
Program TK3TUT15.ASM

TUTORIAL 13:
Command CALL
Command RETURN
Command RETLW
Program TK3TUT16.ASM

TUTORIAL 14:
Tables
Register PCL (again)
Register PCLATH
Program TK3TUT17.ASM

TUTORIAL 15:
Using four switches to create
four different notes
Use of a table to selectively
route program flow
Program TK3TUT18.ASM

TUTORIAL 16:
Indirect addressing
Using unnamed file
locations
Register FSR
Register INDF
Program TK3TUT19.ASM

TUTORIAL 17:
Tone modulation
Command XORLW
Command XORWF
Command IORLW
Command IORWF
Program TK3TUT20.ASM
Program TK3TUT21.ASM
Program TK3TUT22.ASM

TUTORIAL 18:
Register OPTION
Register INTCON
Register TMR0
Use of internal timer
Program TK3TUT23.ASM

TUTORIAL 19:
BCD (Binary Coded
Decimal) counting
Program TK3TUT24.ASM

TUTORIAL 20:
Real-time timing at 1/25th
second
Counting seconds 0 to 60
Program TK3TUT25.ASM

TUTORIAL 21:
Using 7-segment l.e.d.
displays
Showing hours, minutes and
seconds
Command IORLW (usage)
Program TK3TUT26.ASM
Program TK3TUT27.ASM
Program TK3TUT28.ASM

TUTORIAL 22:
Using intelligent l.c.d.s
Setting l.c.d. contrast
Initialising the l.c.d.
Sending a message to the
l.c.d.
Program TK3TUT29.ASM

TUTORIAL 23:
Coding hours, minutes and
seconds for an l.c.d.
Shortened clock monitoring
code
Command SUBLW
Command SUBWF
Program TK3TUT30.ASM

TUTORIAL 24:
Adding time-setting switches
Program TK3TUT31.ASM

TUTORIAL 25:
Writing and reading
EEPROM file data
Register EECON1
Register EECON2
Register EEDATA
Register EEADR
Program TK3TUT32.ASM

TUTORIAL 26:
Illustrating use of EEPROM
data read/write
Converting binary value to
hexadecimal
Program TK3TUT33.ASM

TUTORIAL 27:
Interrupts
Command RETFIE
Program TK3TUT34.ASM
Program TK3TUT35.ASM

TUTORIAL 28:
Command SLEEP
Program TK3TUT36.ASM

TUTORIAL 29:
Watchdog timer (WDT)
Command CLRWDT
Program TK3TUT37.ASM

TUTORIAL 30:
Misc Special Register bits

TUTORIAL 31:
INCLUDE files command
Embedded configuration data
Embedded Data EEPROM values
Embedded PIC type data
Embedded Radix
Program TK3TUT38.ASM

TUTORIAL 32:
PIC16F8x, PIC16F87x,
PIC16F62x family coding
differences
PIC16F87x PORTA
PIC16F87x Data EEPROM use
PIC16F62x PORTA
PIC16F62x Data EEPROM use
Program TK3TUT39.ASM
Program TK3TUT40.ASM

TUTORIAL 33:
Converting binary values to
decimal
Program TK3TUT41.ASM

TUTORIAL 34:
Multiplication routine
Program TK3TUT42.ASM

TUTORIAL 35:
Division routine
Program TK3TUT43.ASM

TUTORIAL 36:
ADC conversion routine for
PIC16F87x family
Program TK3TUT44.ASM

TUTORIAL 37:
CBLOCK command
Interfacing to external serial
EEPROM chips, for PIC16F87x
family
Program TK3TUT45.ASM

TUTORIAL 38:
Outputting serial data at a
specified BAUD rate, for
PIC16F87x family
Program TK3TUT46.ASM

TUTORIAL 39:
Practical example recording
analogue data to serial
EEPROM and subsequent
outputting as RS-232 serial data
Program TK3TUT47.ASM

TUTORIAL 40:
Programming
PICs vs. hardware
Summing-up

APPENDIX A:
Bugged Teaser!

APPENDIX B:
Useful PIC information
Further reading

replied, *"PIC is not an acronym; it is just a trademarked name that General Instruments came up with a long time ago"*. (GI were the originators of PICs.)

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 and peripheral devices would have been required.

PICs are manufactured and supplied "empty". That is, they are without program codes (commands) and cannot control a circuit until they have been provided with a program that tells them what to do. It is the task of the program writer (you) to tell them what that is.

The commands are written in a specialised form of English, largely consisting of mnemonics, known as the "source code". An assembly program (such as *TK3*) then translates (assembles) the source code commands into a numerical form that the PIC can understand, the "program code". This code, which is normally in hexadecimal, is then sent (loaded) in binary format to the PIC by electronic hardware, such as *TK3*'s p.c.b.

PIC VARIETIES

There are many families of PIC microcontroller available, ranging from those which can only be programmed once, to those that can be repeatedly reprogrammed. The former are typically known as One Time Programmable (OTP) devices, and because of this characteristic are not well-suited to hobbyist use since they cannot have their software code changed once they have been programmed.

There are two basic families of reprogrammable PICs: those that require an ultra-violet light unit to erase their previous data before being reprogrammed, but which are now essentially obsolete, and those which are electrically erasable.

In the latter category fall the three device families already mentioned, of which it is the PIC16F84 device we use here. It has been chosen because of its ease of reprogramming and because it does not have additional features that can prove difficult to understand for beginners. Its pinouts are shown in Fig.1, and its basic attributes given in the Specifications panel.

It is an EEPROM (electrically erasable programmable read only memory) device, also known as a "Flash" device, hence the "F" infix in its type number. This means that it can be rapidly reprogrammed as often as you wish, without the need for ultra-violet erasing.

SUB-VERSIONS

Note that there are several sub-versions of individual PIC types, having suffixes such as -04, -10 and -20. The suffix indicates the maximum clock rate at which the chip can be used: 4MHz, 10MHz and 20MHz respectively. You may use any device speed rating for this *Tutorial*, although the -04 is likely to be stocked by more component suppliers.

The PIC16F84 used here has two input/output (I/O) ports, Port A and Port B. Port A has five pins (RA0 to RA4), and Port B has eight pins (RB0 to RB7). We shall be using the PIC in two of its four oscillator modes, RC (resistor/capacitor) and XT (standard crystal up to 4MHz), the former being variable, the latter using a 3·2768MHz crystal.

To re-emphasise an earlier point, 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 PIC16F84 you should have no difficulty applying your knowledge to other PICs.

WHAT YOU NEED

There are six things that you need in order to program a PIC:

● PC-compatible computer having a standard (Centronics-compatible) parallel printer port (USB ports are not suitable)

● purpose built programming hardware board (e.g. *TK3*)

● standard (Centronics) parallel printer port connecting cable

● suitable power supply (*TK3* runs from 9V d.c., which can be supplied via a plug-in mains adaptor)

● word-processing program (text editor)

● assembly and send (download) software program (e.g. *TK3*)

It is worth noting that this *Tutorial* and the *TK3* software can be used with Magenta's version of the *TK3* board, and with the commercial Version 2 PICmicro MCU Development Board. However, these two boards do not have the numbered CP connection points referred to in this text regarding the *EPE TK3* board, but they do have pin function notations and the connections should be obvious.

Data is output from the computer to the PIC via the parallel printer port (addresses 378h, 278h and 3BCh are supported by *TK3*). It is output serially, data on port line D0, and a clock signal on line D1. Additional computer printer port lines are used by *TK3* to enable such functions as reading back program code from a PIC.

You must be able to use a word-processing program. This must produce a text file that is totally without formatting and printer commands. That is, it must be able to generate a pure ASCII text file (and to input one). It is stressed that the source code (.ASM) files *must* be in pure ASCII text formats without printer or display format

TABLE 1. PIC COMMAND CODES FOR PIC16F8x, PIC16F87x AND PIC16F62x

Command/Syntax	Flags affected	Cycles	Description	Tutorial discussed
BYTE-ORIENTED FILE REGISTER OPERATIONS				
ADDWF	f,d	C, DC, Z	1	Add W and f	7
ANDWF	f,d	Z	1	AND W with f	7
CLRF	f	Z	1	Clear f	2
CLRW	–	Z	1	Clear W	2
COMF	f,d	Z	1	Complement f	11
DECF	f,d	Z	1	Decrement f	6
DECFSZ	f,d	–	1 (2)	Decrement f, skip if 0	6
INCF	f,d	Z	1	Increment f	6
INCFSZ	f,d	–	1 (2)	Increment f, skip if 0	6
IORWF	f,d	Z	1	Inclusive OR W with f	17
MOVF	f,d	Z	1	Move f	5
MOVWF	f	–	1	Move W to f	4
NOP	–	–	1	No operation	12
RLF	f,d	C	1	Rotate left f through Carry	4
RRF	f,d	C	1	Rotate right f through Carry	4
SUBWF	f,d	C, DC, Z	1	Subtract W from f	23
SWAPF	f,d	–	1	Swap nibbles in f	11
XORWF	f,d	Z	1	Exclusive OR W with f	17
BIT-ORIENTED REGISTER OPERATIONS				
BCF	f,b	–	1	Bit clear f	2
BSF	f,b	–	1	Bit set f	2
BTFSC	f,b	–	1 (2)	Bit test f, skip if 0	4
BTFSS	f,b	–	1 (2)	Bit test f, skip if 1	4
LITERAL AND CONTROL OPERATIONS				
ADDLW	k	C, DC, Z	1	Add literal and W	7
ANDLW	k	Z	1	AND literal with W	7
CALL	k	–	2	Call subroutine	13
CLRWDT	–	TO, PD	1	Clear Watchdog Timer	29
GOTO	k	–	2	Go to address	1
IORLW	k	Z	1	Inclusive OR literal with W	17, 21
MOVLW	k	–	1	Move literal to W	4
RETFIE	–	–	2	Return from interrupt	27
RETLW	k	–	2	Return with literal in W	13
RETURN	–	–	2	Return from subroutine	13
SLEEP	–	TO, PD	1	Go into standby mode	28
SUBLW	k	C, DC, Z	1	Subtract W from literal	23
XORLW	k	Z	1	Exclusive OR literal with W	17

commands embedded in them. (*TK3* offers a choice of editors, including DOS Edit, Windows Notepad and Windows Wordpad.)

ADDITIONAL COMPONENTS

To use this *Tutorial* with the basic *TK3* p.c.b. you need the following additional components:

330Ω resistor, 0·25W 5% carbon film (8 off)

1µF capacitor, axial elect. 10V

2-line, 16-characters per line alphanumeric liquid crystal display module and datasheet

Optional: 4-digit multiplexed common cathode 7-segment l.e.d. display module and datasheet

Personal (high-impedance) headphones

Jack socket to suit headphones

Extra push-to-make switch for connection via flying leads

Stranded connecting wire

Solder

ANOTHER NEED

Throughout this *Tutorial* we shall examine in a fair amount of detail the 35 basic PIC commands. It is hoped that this will give you all the necessary information that will enable you to conceive a design in which you can use a PIC16F84 to control whatever situation you wish, and to write the code that will let it do so.

There is, though, much more to writing programs than you may at this moment fully appreciate. Knowledge about individual commands and the way in which they can be used is not enough in itself.

Programming is a way of looking at the world that other people may not recognise. You must have the mental ability to see each programming situation as a step by step function, visualising and analysing in your mind exactly how it is that you need to specify the complete program flow.

You have to write the sequence of events with the correct grammar, with the correct spelling and in the correct order. Undoubtedly you will make mistakes while you are writing the code, failing to see the correct sequence of events and using incorrect command structures.

You require the ability to analyse what you have done wrong and to correct it. You are likely to be confronted with an overall task that may, on occasion, take you into several days or even weeks of dedicated concentration.

Readers have occasionally asked how they can be taught to think like a programmer. There is no easy way in which this can be taught. Some people have the ability, some do not. The best way to learn is by actually writing snippets of code and get those to work, giving you the experience and confidence to progress to more complex situations. Throughout this *Tutorial* we try to encourage you in this approach.

Programming, to those who have the ability to see things "as they are" and not "how they seem to be", can become extremely addictive. You could find yourself compelled to get back to the keyboard and PIC programmer at any conceivable hour. You had better have an understanding family!

FIRST THINGS FIRST

To get you started programming PICs through this *Tutorial*, you need the *TK3*

board mentioned earlier plus the few extra components just listed. *TK3* already has eight light emitting diodes (l.e.d.s), four pushbutton switches and four uncommitted (open-collector) *npn* transistors.

These facilities help to illustrate the program examples discussed in the text and encourage you to build up your experience of how the PIC16F84 can be made to respond to different practical situations.

In this text, we start at the very beginning of programming. The first lesson is about the very minimum of information that needs to be written into a program listing before the PIC can do anything else.

We then, "step by step", take a specific very simple task, such as turning on an l.e.d., and describe in detail each of the commands that are required to do the task.

Having described one task, we then take that idea a stage further, adding a few more commands that will enhance the capabilities of the program. Each of these commands is similarly discussed in detail. Thus we progress, taking simple ideas, and illustrating how they can be achieved and then enhanced.

We feel that this approach is far more useful than describing each and every one of the commands in turn before we ever get to use them. Most people learn by doing, reading about things in short sections and applying the knowledge in practical bite-sized chunks. A complete chapter on all the commands in sequence would be too much to remember and understand in one go.

The complete list of Microchip's commands for the PIC16F8x, PIC16F87x and PIC16F62x families is shown in Table 1. All are discussed and demonstrated. In the early years of PICs there used to be two others, OPTION and TRIS, but Microchip have dropped them from their recent PIC families.

As you will see, there are a lot of "bit-orientated" sub-commands as well as the main commands. Some of these are similar in operation and close examination will be given only to the principal ones – once you understand these, use of the similar bit-setting commands available will become obvious.

TUTORIAL SECTIONS

The *Tutorial* is split into numbered sections. Each deals with specific coding topics but, in most cases, is a direct follow-on from the previous one, and is nearly always visually illustrated by the displays on the *TK3* p.c.b. The exception to the latter is when sound is used as the output medium, when the personal headphones are needed.

At the end of most sections there are a few simple exercises which allow you to experiment with the program presented in that section. You will only be expected to use the commands that have already been introduced to you. None of them should tax your brain too much, but they will, hopefully, encourage you to think of alternative ways in which the same basic task can be tackled, or to consider other tasks that can be achieved using similar techniques.

By the end of the complete *Tutorial*, you will know how to get the PIC16F84 to respond to switches and other external signal sources, send data to various types of display, to create sound, to be the heart of a 24-hour clock and to store data in its non-volatile EEPROM memory.

Readers who have had experience of programming in BASIC, or with other

types of microprocessor or microcontroller, will find that once a few commands have had their functions explained, using them will rapidly become instinctive. The author, having many years of such experience, effectively learned about PICs and how to use them over a single weekend, just by doing a bit of experimenting.

Other readers without such experience will, it has to be said, have to become accustomed to understanding programming itself as a step by step process. An analytical mind is required and, as said earlier, there is no easy way in which programming can be taught to those who lack this ability.

TUTORIAL EXERCISES

Throughout this *Tutorial* we present various programming exercises at which to try your hand. In most cases, you are requested to modify an existing tutorial program. This requires it to be saved, assembled and sent to the PIC.

It is important that when you make these changes, you do not save the variant under the same name as the original. It must have a different name. If you *do* save under the same name, the original file will be replaced by the new one. You can save each program variant by any name of your choosing, but use the same .ASM extension as the examples use. You might consider using TRY1.ASM, TRY2.ASM, etc.

It is recommended that you save the original file under the new name before you make any changes, to avoid curse-worthy errors! But if you do make a mistake and overwrite something you should not, the original can be recopied from its disk or FTP site download.

Having saved your changed file, you now assemble and send it to the PIC.

We shall *not* be giving possible solutions (of which there could be several) to any of the exercise questions. It is expected that you will persevere until you find a workable solution. Only in this way will you get into the habit of being presented with a computing problem, which you have to solve on your own, and then solving it.

This last statement was made in the original *Tutorial* text but we still periodically get asked what the solutions are. We are hard-hearted on this point and don't offer solutions! If you want to become a programmer, you've got to get your brain thinking like one. The exercises are all simple and have simple solutions, we would not be doing you any favour by telling you some answers.

We are now almost in a position to start telling you about writing programs for a PIC16F84 and for you to start getting your brain into gear – it's really all very logical!

PREPARATION

From hereon you need the *TK3* software running on your PC. It should be loaded and run as explained in its published text. You also need the *TK3* p.c.b. connected to the PC via its parallel printer port cable, and a PIC16F84 in the allocated socket. You do not need the l.c.d. or 7-segment l.e.d. modules connected at this time. However, if the l.c.d. module is already connected you do not need to remove it.

All the software for this *Tutorial* should be in the same folder. This may be the same folder as used for *TK3*'s software, or in
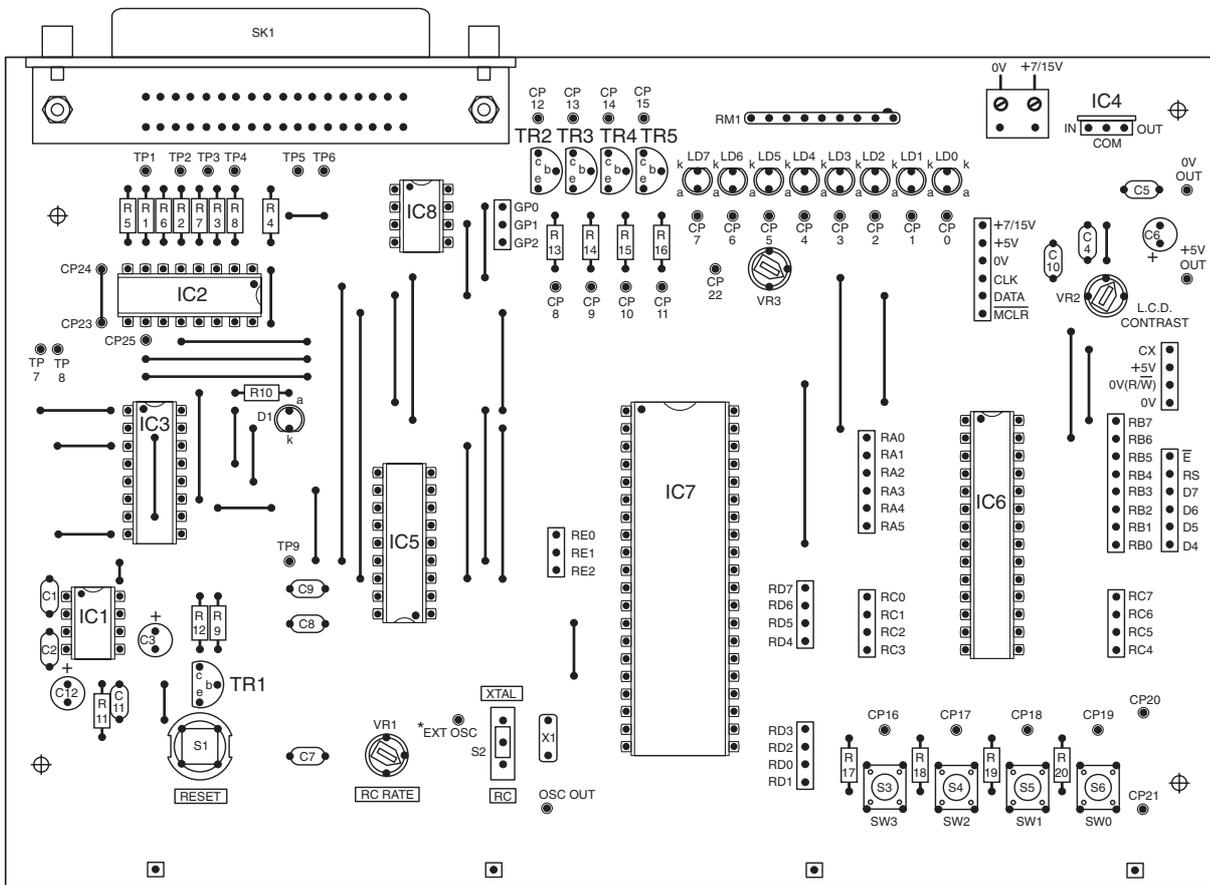
Fig.2. Component layout for the TK3 board to which you should refer for the connection points required for the exercises.
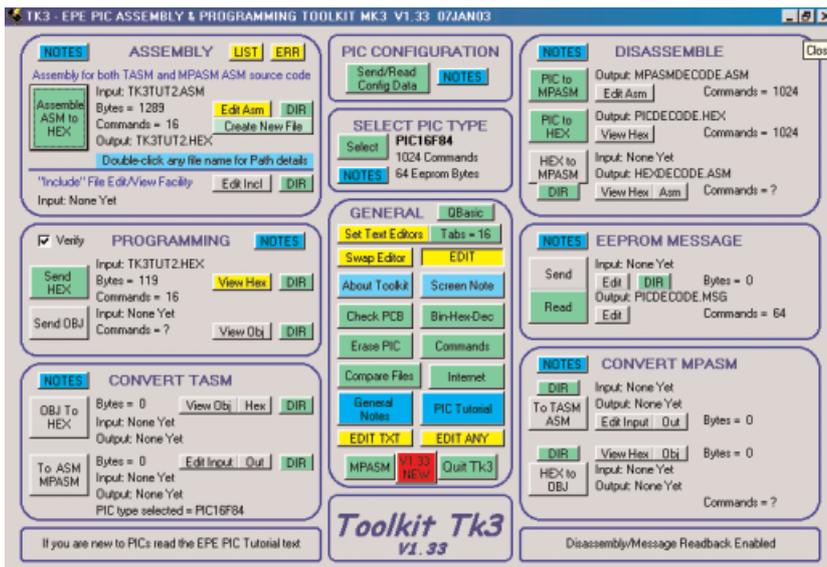
Main screen of TK3's software suite.
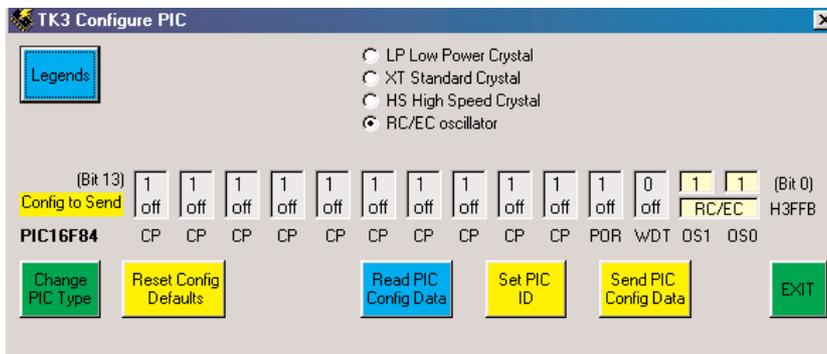
Configuration settings used for the first several experiments, putting the PIC16F84 into RC oscillator mode.

another folder having any name of your choosing. There is no "installation" required, just copy the files into this folder from your disk or FTP site download, using Windows' own copying facilities.

As we progress, you need to connect various PIC pins to such on-board items as l.e.d.s and switches. This will be explained as required. In the meantime, keep your soldering iron switched on and ready! Remember that you should always switch off *TK3*'s power *before* making any soldered changes. It is also recommended that the printer port cable should be disconnected at this time as well.

The first few Tutorials require PIC Port B to control the eight l.e.d.s (LD0 to LD7) on *TK3*'s board (see Fig.2). Solder short lengths of wire between the allocated RB0/RB7 pins and the CP0/CP7 pins, strictly observing the same numerical order – RB0 to CP0, RB1 to CP1 etc.

Additionally, solder a 1μF capacitor across (or in place of) the existing capacitor C7 (unless C7 is already a 1μF device, in which case leave it in position), its positive lead facing preset VR1. This reduces the oscillator rate to a very slow range.

Also for the sake of these Tutorials, put a sticky label in front of switches S3 to S6 and label them SW3, SW2, SW1, SW0, in that order from left to right.

PIC CONFIGURATION

The first thing to understand is that all PICs must be "configured" for the application they are intended to control. Such configuration includes the selection of oscillator type, and matters such as Watchdog timer and Code Protection bits use (which are discussed in Part 3). Once the configuration has been sent it is not normally necessary to change it for the same application.

In this series the setting of the configuration data into the PIC is treated as one programming operation. Sending the program itself is then another. As discussed in Part 3, it is also possible with some programmers (such as *TK3*) to "embed" the configuration code into the program itself. When the program is sent to the PIC, the configuration is then automatically set as required.

Note that occasionally (and for a variety of reasons) the configuration in the PIC can become corrupted. If you find that a program which you believe should run does not, run the configuration program (about to be described) to ensure that the PIC is correctly configured, and then re-send the program.

Sometimes corrupted configuration can prevent another program from replacing an existing one, the latter continuing to run even though the new one has been sent. In this case, use *TK3*'s Erase PIC option, and failing that run its Clear Code Protection routine (both functions described in *TK3*'s text).

Be aware that the time taken to send a program (.HEX) file to the PIC depends on its length. A delay exists because the PIC requires a minimum amount of time to store each command when received. The speed of operation has nothing to do with the speed of your computer.

Before starting the Tutorials and their exercises, configure your PIC16F84 via *TK3*. Click on the Send/Read Config Data button on *TK3*'s main screen (top centre – PIC Configuration panel). Set the configuration for RC (resistor/capacitor oscillator) with all other functions off. Send the data to the PIC. Always keep Code Protection off throughout these Tutorials.

If your PIC has been used before in some other application, you should also clear its existing program and data EEPROM contents via *TK3*'s Erase PIC button.

As the PIC is configured for RC operation, *TK3*'s RC/XTAL slide switch (S2) also needs setting for RC. Set the RC control preset VR1 for slowest oscillator rate (fully anti-clockwise).

(If you are not clear about any aspect relating to *TK3*, re-read its published text.)

NUMERICAL PREFIXES

There are four formats that can be used to express numerical values in a program. They may be expressed in decimal (e.g. 0, 4, 5), or in hexadecimal (indicated by a prefix of H', e.g. H'00', H'04' and H'05' in this case), or in binary (indicated by a prefix of B', e.g. B'00000000', B'00000100' and B'00000101'). Prefixed values must be concluded with an apostrophe suffix (') – as used in the prefix. On your keyboard the apostrophe required is that which is marked below the @ symbol (the one at top left by the numerals is not suitable).

Be aware that some PIC assembly programs (but not *TK3*) may require decimal numbers to be prefixed with D', e.g. D'10' or D'255'. It is also worth noting that some assembly programs (including *TK3*) accept the use of the dollar sign ($) to indicate hex, e.g. $F7 instead of H'F7', and for binary numbers to be prefixed by a percentage sign (%), e.g. %01010101 instead of B'01010101'.

Check with any other assembly program's documentation before using the shorter method. The D', B' and H' prefixes

---

LISTING 1: PROGRAM TK3TUT1

```
; TK3TUT1.ASM
; minimum requirement

   ORG 0        ; Reset Vector address
   GOTO 5       ; go to PIC address
                  location 5
   ORG 4        ; Interrupt Vector address
   GOTO 5       ; go to PIC address
                  location 5
   ORG 5        ; Start of Program
                  Memory

; (your program goes in here)

   END          ; final statement
```

---

are those used with Microchip's own MPASM assembly programs and they do not accept % or $ prefixes. The B' and H' prefixes are used throughout these Tutorials. *TK3* accepts both prefix formats.

You will find *TK3*'s Binary to Hex to Decimal conversion option useful if you wish to convert between the three numerical formats.

TUTORIAL 1
CONCEPTS EXAMINED

Minimum commands needed
Port default values
Instruction ORG
Instruction END
Command GOTO

The absolute bare minimum requirements for any PIC program that is to be assembled (compiled) are shown in Listing 1.

In fact, none of the statements in this listing have anything directly to do with a functioning software program. Six are aimed directly at the software assembly program, the others are comments to the human programmer, or other reader. Such comments include program title and function, and notes about what tasks particular program instructions within the list are intended to perform.

Comments must always be preceded by a semicolon (;) so that the assembler does not try to treat them as program commands. 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:

; TK3TUT1.ASM
; minimum requirement

Next come five commands which are aimed at the software assembly program (e.g. *TK3*) as well as the PIC. They need not normally concern you, repeating the commands parrot-fashion in any software you write will normally suffice (unless interrupts are involved – discussed in Part 3).

The three ORG (origin) commands and their associated address (program memory location) values tell the assembly program

---

at which memory address within the PIC a particular set of subsequent commands is to be placed.

Position ORG 0 is known as the Reset Vector. It is to this address that the PIC jumps when it is first run or subsequently reset.

Position ORG 4 is known as the Interrupt Vector. It is to this address that PIC jumps if an interrupt occurs. The subject of interrupts will be dealt with in Tutorial 27. Ignore the concept for the moment.

Position ORG 5 is the Start of Program Memory, i.e. it is the first available position within the PIC at which the actual program itself can start.

The first two ORG statements have to be followed by a GOTO command (the first of the recommended 35 commands that the PIC understands and which you need to know!), plus an address value. The GOTO command (not surprisingly) simply tells the PIC to GO TO the address stated. The addresses can be any chosen by the program writer, but in these Tutorials are taken as GOTO 5, address 5 being the Start of Program Memory, as indicated by the ORG 5 statement.

You will notice that locations 1, 2 and 3 are not mentioned. These are reserved by the PIC and are not available for normal program use.

The bracketed statement in the listing following ORG 5 is aimed at you, the program writer: 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 only required by some assembly programs. With *TK3* it is not essential, but you should always include it at the very end of any listing in case your programs are assembled by software that does require it.

Having included the essential first five commands and the END statement, everything else beyond ORG 5 is up to you.

INCAPABLE?

You might think that when TK3TUT1.ASM is assembled and loaded into the PIC, the PIC will be incapable of doing anything – it hasn't been told of anything to do, other than GOTO 5. Almost true, but not quite!

PICs have been told in manufacture to adopt certain "default" conditions when first switched on (those for the PIC16F84 will be shown in Part 3). One of these default conditions is that Port A and Port B are configured (set) to act as inputs. In this condition they are simply held in a high-impedance state. What is not configured at this time is the binary value which is available to be output via those pins when they are first set as outputs from within the program.

At switch-on, any number could be set randomly within the PIC's memory, of between 0 and 255 (B'00000000' to B'11111111') for Port B (eight pins), and 0 to 31 (B'00000' to B'11111') for Port A (five pins). It is often preferable, therefore, to set port output values to a known value as part of the opening program statements. This, too will become apparent as we progress.

It is also worth noting that PIC pins should never be left as "floating" inputs. If

LISTING 2 – PROGRAM TK3TUT2

```
; TK3TUT2.ASM
; setting Port B to output mode and turn
on each l.e.d.
    ORG 0      ; Reset Vector address
    GOTO 5     ; go to PIC address 5
    ORG 4      ; Interrupt Vector address
    GOTO 5     ; go to PIC address 5
    ORG 5      ; Start of Program Memory
    CLRF 6     ; set all Port B pins to logic 0
    BSF 3,5    ; instruct program that a
                 Bank 1 command comes
                 next
    CLRF 6     ; set all Port B pins as
                 outputs
    BCF 3,5    ; instruct program that a
                 Bank 0 command comes
                 next
    BSF 6,0    ; set Port B pin 0 to logic 1
    BSF 6,1    ; set Port B pin 1 to logic 1
    BSF 6,2    ; set Port B pin 2 to logic 1
    BSF 6,3    ; set Port B pin 3 to logic 1
    BSF 6,4    ; set Port B pin 4 to logic 1
    BSF 6,5    ; set Port B pin 5 to logic 1
    BSF 6,6    ; set Port B pin 6 to logic 1
    BSF 6,7    ; set Port B pin 7 to logic 1
    END        ; final statement
```

any PIC pins remain unused in a PIC-controlled circuit, they should either be biased to one or other power line by individual resistors (say 10kΩ to 100kΩ), or set as outputs in a logic 0 (low) condition.

There are no exercises for Tutorial 1.

TUTORIAL 2
CONCEPTS EXAMINED

Clock cycles
File registers
Bits
Bytes
Set
Clear
Command CLRF
Command CLRW
Command BSF
Command BCF
Ports and Port directions
Register STATUS
STATUS register bit 5
Banks 0 and 1

CONNECTIONS NEEDED

All Port B to all l.e.d.s.
Capacitor C7 as 1μF
Preset VR1 set to maximum resistance (fully anti-clockwise)

At this point in time, a PIC without program commands is of no benefit to us! We shall now demonstrate a simple program which just turns on a series of eight l.e.d.s connected to Port B. Look at Listing 2.

As discussed in Tutorial 1, you will see comments at the start of the listing and within it, all preceded by a semicolon. You will also see the ORG and END statements, plus the GOTO 5 commands. Sandwiched between ORG 5 and END are several lines of code. The PIC considers the first program command (CLRF 6) as being at its address location 5, which is where the assembly program will place it when it loads the code into the PIC.

Ignoring the numbers following the commands, there are only three different commands in use in this program section (routine): CLRF, BSF and BCF. They simply stand for CLeaR File, Bit Set File and Bit Clear File. An allied command to CLRF is CLRW (CLeaR Working register).

Via *TK3*, assemble the program TK3TUT2.ASM, and load the resulting TK3TUT2.HEX code into the PIC. Having loaded it, the program automatically starts running. There will be a few seconds wait before you see anything happening.

From here on, whenever you are asked to load or run a program, it must first be assembled, and then its HEX code loaded into the PIC. It will automatically run when loading has finished.

At present, the oscillator which controls the PIC is only running at a very slow speed, about 1Hz or so (depending on the tolerance of the C7 and VR1 values). The internal workings of the PIC16F84 (and other PIC types) automatically divide any clock frequency by four, taking a minimum of four counts to process a single command. During each of the four intervening pulses, different aspects of the command are processed. To all intents and purposes, each command takes four clock pulses.

Each batch of four clock pulses is known as a *clock cycle*. Most commands take just one clock cycle, although some take two, partly depending on conditions resulting from their operation (see Table 1). The GOTO command takes two cycles to complete, whereas BSF, BCF, CLRF and CLRW take just one cycle.

The results of the first six commands will not produce any visible result. After a few seconds, though, each l.e.d. connected to Port B will come on in turn, in order of LD0 to LD7, with a pause between each change. When the final l.e.d. (LD7) has come on, there are no more instructions to perform, so nothing more happens (in fact, the PIC is actually working its way through each of its locations that have not been provided with code by the program).

To see the sequence again, press *TK3*'s Reset switch (S1), whereupon the program will restart from the beginning. The rate at which it occurs can be changed by adjusting preset VR1.

(Note that if the l.c.d. module is already connected, a very slight glow from some l.e.d.s. may just be visible, and the top line of the l.c.d. will contained darkened cells – this is normal.)

But, how does the program do what you see it has done? First, we'll tell you what File registers are and what commands CLRF, BSF and BCF do.

TABLE 3. PIC MEMORY CAPACITY

Device Type	Program size	Data EEPROM	Registers	Pins
PIC16F627	1024	128	224	18
PIC16F628	2048	128	224	18
PIC16F83	512	64	36	18
PIC16C84	1024	64	36	18
PIC16F84	1024	64	68	18
PIC16F84A	1024	64	68	18
PIC16F870	2048	64	128	40
PIC16F871	2048	64	128	40
PIC16F872	2048	64	128	28
PIC16F873	4096	128	198	28
PIC16F874	4096	128	192	40
PIC16F876	8192	256	368	28
PIC16F877	8192	256	368	40

TABLE 2. PIC16F84 REGISTER FILE MAP (courtesy Microchip)

File Address			File Address
00h	Indirect addr.(1)	Indirect addr.(1)	80h
01h	TMR0	OPTION	81h
02h	PCL	PCL	82h
03h	STATUS	STATUS	83h
04h	FSR	FSR	84h
05h	PORTA	TRISA	85h
06h	PORTB	TRISB	86h
07h			87h
08h	EEDATA	EECON1	88h
09h	EEADR	EECON2(1)	89h
0Ah	PCLATH	PCLATH	8Ah
0Bh	INTCON	INTCON	8Bh
0Ch			8Ch
	68 General Purpose registers (SRAM)	Mapped (accesses) in Bank 0	
4Fh			CFh
50h			D0h
7Fh	Bank 0	Bank 1	FFh

☐ Unimplemented data memory location; read as '0'.

Note 1: Not a physical register.

The named file addresses are known as the Special Function Registers.

FILE REGISTERS

The PIC16F84, and the other PICs mentioned earlier, have five areas of memory (see Table 3 for quantities):

1. Program Memory (EEPROM) in which are stored the commands that form the program, and where the program of TK3TUT2 is now held.

2. Data Memory (SRAM – static random access memory) in which you can *temporarily* store the results of any action that the program performs (the data is lost when the power is switched off).

3. Data Memory (EEPROM) in which you can *indefinitely* store any data that you wish to retain after power has been switched off (it will be discussed in Tutorial 25).

4. Special Function Memory (SRAM), whose attributes determine what actions the PIC takes in respect of program commands (Table 2).

5. Working Memory (SRAM – 1 byte), through which many operations have to pass during program performance (in other processors the Working Memory may be called the Accumulator).

The results (data) of any program action can be directed to be stored at any of the memory areas numbered 2 to 5 above. With items 2, 3 and 4 the destination is known as a File destination. With item 5, the Working Memory or Register, the destination is known as, not surprisingly, the Working destination.

Strictly speaking, Files should be known by their full name of File Registers. It would be tedious, though, to keep using the full name, and so the term "file" will be used throughout the Tutorials to mean File Register. (We also use the term "file" in relation to disk files – the context should make the meaning clear.)

For those of you who are familiar with programming in BASIC, files can be regarded as the equivalent of variables. There is no direct BASIC equivalent of the Working register, though in a sense it can be regarded as a special variable.

The program commands reflect the file and working destinations by the use of F and W, respectively, in the code itself. For example, in the code CLRF, the F indicates that the value in a particular file (memory data byte) is to be CLeaRed (reset to zero) and the result is to be retained in the file. On the other hand, in the code MOVLW (to be met in Tutorial 4) the W stands for Working (in this case meaning that a Literal value is to be MOVed into W). The use of both F and W in a command (e.g. MOVWF) will become evident in due course.

BITS

Having established what a file is, it is necessary (and easy) to understand the concept of a "bit". You no doubt understand it already, but, just to recap, a bit is a single part of an electronic memory which can be set to one of two states: either to logic 1 ("on" – charged to a voltage which is usually the same as the positive power rail that supplies the circuit, e.g. +5V); or to logic 0 ("off" – discharged to a zero voltage). Logic 1 and logic 0 are often referred to as high and low, respectively.

A memory i.c. can have any number of bits contained within it. A fixed number of bits is known as a "byte". There is a common misconception that a byte is only ever comprised of eight bits. Historically, this is not true, any fixed number of bits which can be operated as a single unit is called a byte. However, by current usage, a byte is usually taken to be comprised of eight bits.

The status of the bits (high or low) within a byte is expressed as a binary number reading from left to right, in order of the bit representing the highest value first (most significant bit – MSB) down to the lowest value (least significant bit – LSB). Thus decimal 128 is expressed as binary 10000000, whereas binary 1 may be expressed as 00000001 (the preceding number of zeros may not always be included – their presence being implied).

The bits of a byte are referred to by their position within a byte, with position 0 at the right, ascending as high as necessary depending on the byte length. Thus an 8-bit byte has its bits numbered as 7, 6, 5, 4, 3, 2, 1, 0. A 16-bit byte (probably referred to as a *word*) would be numbered from 15 to 0. The use of 0 (zero) as a bit number is essential to remember when programming PICs.

Incidentally, terms MSB and LSB can also be used to mean Most Significant Byte and Least Significant Byte. The appropriate meaning should be clear from the context. There are also similar terms, NSB, NMSB and NLSB, in which the N stands for Next.

TABLE 4: STATUS REGISTER (Courtesy MICROCHIP)

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C
bit7							bit0

R = Readable bit
W = Writable bit
U = Unimplemented bit, read as '0'
- n = Value at POR reset

bit 7: **IRP**: Register Bank Select bit (used for indirect addressing)
0 = Bank 0, 1 (00h - FFh)
1 = Bank 2, 3 (100h - 1FFh)
The IRP bit is not used by the PIC16F8X. IRP should be maintained clear.

bit 6-5: **RP1:RP0**: Register Bank Select bits (used for direct addressing)
00 = Bank 0 (00h - 7Fh)
01 = Bank 1 (80h - FFh)
10 = Bank 2 (100h - 17Fh)
11 = Bank 3 (180h - 1FFh)
Each bank is 128 bytes. Only bit RP0 is used by the PIC16F8X. RP1 should be maintained clear.

bit 4: **$\overline{TO}$**: Time-out bit
1 = After power-up, CLRWDT instruction, or SLEEP instruction
0 = A WDT time-out occurred

bit 3: **$\overline{PD}$**: Power-down bit
1 = After power-up or by the CLRWDT instruction
0 = By execution of the SLEEP instruction

bit 2: **Z**: Zero bit
1 = The result of an arithmetic or logic operation is zero
0 = The result of an arithmetic or logic operation is not zero

bit 1: **DC**: Digit carry/borrow bit (for ADDWF and ADDLW instructions) (For borrow the polarity is reversed)
1 = A carry-out from the 4th low order bit of the result occurred
0 = No carry-out from the 4th low order bit of the result

bit 0: **C**: Carry/borrow bit (for ADDWF and ADDLW instructions)
1 = A carry-out from the most significant bit of the result occurred
0 = No carry-out from the most significant bit of the result occurred
Note: For borrow the polarity is reversed. A subtraction is executed by adding the two's complement of the second operand. For rotate (RRF, RLF) instructions, this bit is loaded with either the high or low order bit of the source register.

TERMS SET AND CLEAR

The concept of the terms "set" and "clear" is important to understand. In program terms, to "set" a bit means to force it high, i.e. to logic 1; the term "clear" is used to mean that a bit is forced low, i.e. to logic 0.

Note, however, that in textual terms (i.e. in articles such as this) you are likely to come across the mixed use of the word "set", in that you might be told to "set a bit low". In such cases, the implied meaning should be obvious from the context. In this example, "low" is the important word and "force" or "make" could have been used instead of "set".

COMMANDS CLRF AND CLRW

Command CLRF stands for CLeaR File and is always followed by a single number or name which indicates the file on which the action is to be performed. The command instructs the PIC that all bits within the stated file are to be cleared, i.e. it clears the whole byte (to hold a value of zero).

The allied command CLRW (CLeaR Working register) simply clears the contents of the Working register and is used on its own without a subsequent number or name. In practice, this command may be seldom used, the commands MOVLW 0 and RETLW 0 probably finding preference (these commands are discussed in Tutorials 4 and 13, respectively). Command CLRW has, in fact, been dropped from some PIC families.

There are no direct opposites of CLRF and CLRW which will set all bits high; other techniques have to be used for this action.

Using names for files instead of numbers will be dealt with in Tutorial 3.

COMMAND BSF

PICs can have any individual bit of any file byte acted upon directly. Each bit can be set high or low by a single command, and, as we shall show later, a single command will also determine the status of any individual bit.

Command BSF translates as Bit Set File and is always followed by two numbers or names separated by a comma. The first number or name is the file byte whose bit is to be acted upon. The second number or name is that representing the number of the bit within the file byte (between 7 and 0, reading from left to right).

As an example, the command BSF 3,5 in Listing 2 instructs the PIC to set (make high, force to logic 1!) bit 5 of file number 3; command BSF 6,7 means that bit 7 of file number 6 is to be set.

COMMAND BCF

Command BCF stands for Bit Clear File and is the exact opposite of BSF. As an example, the command BCF 3,5 in Listing 2 means that bit 5 of file 3 is to be cleared (set low, made/forced to logic 0!).

PORT FILE NUMBERS

In the example of Listing 2, the files whose bits have been set or cleared are those which control Port B. In terms of writing to or reading from Port A and Port B, the two ports are treated as any other file destination. However, Port A and Port B are different to other files in that they are for communication with the outside world.

There are, in fact, two file registers associated with each port that are accessible to the programmer. One is used to set the direction in which the bits are to act, i.e. as inputs or outputs (it is also known as the Data Direction Register – DDR), and the other deals with the data written to (for output to the world) or read from (input from the world). Each bit of the port direction registers can be individually set or cleared so that the same port may be used simultaneously for data input and output via different bits.

The file which controls the direction in which Port B pins respond is named in the PIC datasheets as TRISB. It is one of the named "Special Function Register" files (see Table 2) and is contained in memory byte address H'86' (we won't translate hexadecimal numbers into decimal since the latter are irrelevant in the context of file addresses).

The file which holds Port B's data, whether as input or output, is at memory byte address H'06' and, helpfully, is known in the PIC datasheet as PORTB.

A minor inconvenience exists in the PIC16F84, and the other PIC families referred to earlier, in that addresses H'80' to H'8B' can only be accessed by changing the value in another file, the STATUS file (Table 4). This file is held jointly at byte addresses H'03' and H'83'. A single bit within STATUS is used to direct the program to numbers either below H'80' (known as Bank 0 or Page 0 addresses) or above H'7F' (known as Bank 1 or Page 1 addresses); this bit is number 5. (Note that some PICs have more than two Banks.)

When STATUS bit 5 is set (logic 1), it effectively adds H'80' to the memory byte address being accessed. When the bit is clear (logic 0), addresses below H'80' are accessed. Thus, if you instruct that file 6 (H'06') is to be accessed when STATUS bit 5 is low, the file actually at address 6 will be accessed. Conversely, if you instruct that file 6 is the subject when STATUS bit 5 is high, the file at address H'86' will be accessed.

In the PIC16F84, only files numbered H'00' to H'4F' and H'80' to H'8B' are available for use by the programmer. Note that files H'07' and H'87' have no function.

It may appear from the Registers File Map in Table 2 that addresses H'8C' to H'CF' might also be available. Writing to these addresses, though, simply accesses those between H'0C' and H'4F'. (In some PICS, including the PIC16F87x and PIC16F62x families, additional memory is available at the higher Bank addresses – see *Using the PIC16F87x Additional Memory, EPE* June '02, on the CD-ROM.)

LISTING 2 COMMANDS

We can now examine each command of Listing 2 in turn and describe its purpose. When the PIC is first switched on, the STATUS file is set to a default value with its bit 5 low. All file addresses are thus treated as being below H'80'.

We have established that address 6 is that which holds the data for Port B. The first command, CLRF 6, thus clears the data which is held in Port B as a value available to be output, i.e. Port B's output register is instructed to hold a value of zero.

The purpose of the program in Listing 2 is to output data to the eight l.e.d.s on Port B, and it has already been said that the default value of Port B's direction register (at H'86') is for all bits to be set for input (all bits are high – 11111111). Consequently we must now set them all as outputs, i.e. each to logic 0, thus 00000000. To do this, first the STATUS register at address 3 must have its bit 5 set high to point to addresses of H'80' and above; hence the command BSF 3,5.

Now we configure all of Port B for output mode with the command CLRF 6. Yes,

it's the same command as cleared Port B's data, but because STATUS bit 5 is high, the value of 6 (H'06') has H'80' added to it, so the address actually accessed is that at H'86'.

The commands which output data to the l.e.d.s are all concerned with Port B's data file at "real" address 6, so the addition of H'80' is no longer needed. The next command, BCF 3,5, thus clears bit 5 of the STATUS register at address 3. All remaining commands in Listing 2 can now, in turn, set high each data bit of Port B at address 6: BSF 6,0, BSF 6,1, etc., so turning on the l.e.d.s in sequence from LD0 to LD7. Simple!

EXERCISE 2

2.1. Using your text editor and a *copy* of TK3TUT2.ASM (renamed to any title of your choosing, but still with the .ASM extension), experiment with the eight commands relating to file 6, using different values (between 0 and 7) for the number following the comma. Do not change the number before the comma. Also experiment with changing BSF to BCF.

2.2. Rewrite the program in Listing 2 so that it performs its actions on Port A instead of Port B. The equivalent data and direction addresses for Port A are 5 (H'05') and H'85', respectively. Note that Port A

LISTING 3 – PROGRAM TK3TUT3

```
; TK3TUT3.ASM
; using names to ease writing of Listing 2

STATUS     EQU 3                      ; name program location 3 as STATUS
PORTB      EQU 6                      ; name program location 6 as PORTB

           ORG 0                      ; Reset Vector address
           GOTO 5                     ; go to PIC address location 5
           ORG 4                      ; Interrupt Vector address
           GOTO 5                     ; go to PIC address location 5
           ORG 5                      ; Start of Program Memory

           CLRF PORTB                 ; clear Port B data pins
           BSF STATUS,5               ; set for Bank 1
           CLRF PORTB                 ; set all Port B as output
           BCF STATUS,5               ; set for Bank 0

LOOPIT     BSF PORTB,0                ; set Port B pin 0 to logic 1
           BSF PORTB,1                ; set Port B pin 1 to logic 1
           BSF PORTB,2                ; set Port B pin 2 to logic 1
           BSF PORTB,3                ; set Port B pin 3 to logic 1
           BSF PORTB,4                ; set Port B pin 4 to logic 1
           BSF PORTB,5                ; set Port B pin 5 to logic 1
           BSF PORTB,6                ; set Port B pin 6 to logic 1
           BSF PORTB,7                ; set Port B pin 7 to logic 1
           CLRF PORTB                 ; clear all PORTB pins
           GOTO LOOPIT                ; go to address LOOPIT
           END                        ; final statement
```

LISTING 3A (TK3TUT3 LIST FILE)

List count deci	Prog count deci	Prog count hex	Code value hex	Source code	
0004	0	0000		STATUS	EQU 3
0005	0	0000		PORTB	EQU 6
0006	0	0000			
0007	0	0000			ORG 0
0008	0	0000	28 05		GOTO 5
0009	1	0001			ORG 4
0010	4	0004	28 05		GOTO 5
0011	5	0005			ORG 5
0012	5	0005			
0013	5	0005	01 86		CLRF PORTB
0014	6	0006	16 83		BSF STATUS,5
0015	7	0007	01 86		CLRF PORTB
0016	8	0008	12 83		BCF STATUS,5
0017	9	0009			
0018	9	0009	14 06	LOOPIT	BSF PORTB,0
0019	10	000A	14 86		BSF PORTB,1
0020	11	000B	15 06		BSF PORTB,2
0021	12	000C	15 86		BSF PORTB,3
0022	13	000D	16 06		BSF PORTB,4
0023	14	000E	16 86		BSF PORTB,5
0024	15	000F	17 06		BSF PORTB,6
0025	16	0010	17 86		BSF PORTB,7
0026	17	0011	01 86		CLRF PORTB
0027	18	0012	28 09		GOTO LOOPIT

The full listing also shows binary values and comment statements.

only has five pins, not eight. What difference, if any, does this make to the program? You need to disconnect the wires from Port B that go to l.e.d.s LD0 to LD4, and then connect Port A pins to these l.e.d.s in correct numerical order.

You will notice that Port A pin RA4 does not appear to turn on its l.e.d. This is because the pin has an "open-collector" output which needs to be biased high, before it can be used to toggle between Logic 0 and Logic 1. An example of this is shown in Tutorial 12.

Reinstate the Port B connections to all l.e.d.s when you have finished with Exercise 2.2.

TUTORIAL 3
CONCEPTS EXAMINED

Names in place of numbers
Labels
Case sensitivity
A repetitive loop
Instruction EQU

CONNECTIONS NEEDED
All Port B to all l.e.d.s.
Capacitor C7 as 1μF
Preset VR1 set to maximum resistance (fully anti-clockwise)

In the previous section dealing with Listing 2, we were using numbers to indicate which file was being referred to. That's fine if there are only a few files whose address numbers can be easily remembered. As we progress with examining the PIC commands and example listings, though, we shall be using more and more files. If we continue to refer to them numerically, we're going to get lost! (What's file H'0C' for? Is the data supposed to go to file H'1C' or file H'1E'?)

Human memories with numbers are notoriously bad! But we are (usually) much better with names. This fact was long ago recognised by program writers and many types of software allow the use of names in place of numbers. Let's examine how names can be applied to the numbered files in Listing 2. Have a look at Listing 3.

Any number written into a listing can be represented by a name. It can be any name you like as long as you think you'll know in time to come what is meant by that name (but some assembly programs impose a limit on name lengths, although *TK3* does not). There are also some names which it is better to allocate according to their function, especially those functions that already have names provided in the PIC datasheets, such names as STATUS and PORTB, for example.

With some programmers (but not *TK3*), the names are "case-sensitive". In other words, once you have equated a name with a number, further use of the name must be in exactly the same style as the original with regard to the use of upper and lower case letters. For example, names PORTB and portb should not be used interchangeably. It is recommended, though, that you do not use different upper and lower case styles of the same word to mean different things.

However, the commands themselves (as opposed to the names) may be in upper or lower case without (usually) causing problems. For example, for clarity on these published pages, the commands are shown in upper case. In the actual full ASM listings, though, the same commands are principally in lower case – the *TK3* assembly program recognises both styles.

EQUATING NAMES
In Listing 3, three names have appeared: the aforementioned "aliases" STATUS and PORTB, plus LOOPIT. We'll keep LOOP-IT a mystery for the moment (but we're sure you know what it's for). All "alias" name allocations must appear at the head of the program listing, in the initialisation block.

Then the format for allocating a name to a number is to state the name at the left of a program line followed by at least one space, although you may have more than one space if you prefer, to keep things looking neat and tabulated. Most assembly programs allow the use of the Tab key to keep columns tabulated, which makes typing easier than keying-in lots of spaces.

Now the statement EQU is made, followed by a space and the number you wish to name. In the case of the first line, STATUS is the name to be given to the numeral 3 (which, you will remember, is the file address number for the STATUS register). Hence, the statement:

STATUS EQU 3

This simply tells the assembler program that when it assembles the listing into code, each time it comes across the name STATUS, it is to replace it in the code by the value of decimal 3.

The second line similarly allocates the name PORTB to file address 6:

PORTB EQU 6

Compare Listing 2 and Listing 3; you will see how the program has been rewritten using the names in place of file numbers.

It is permissible to use other numerical formats, such as binary and hexadecimal, in EQU number defining.

LIST (.LST) FILES
When ASM source code files are assembled, a List file (.LST) is generated as well as the HEX file. A list file allows examination of the original ASM source code and the actual values that the assembly program generates in respect of that code.

Some assemblers generate separate list files for each source code file, using the same basic file name, but giving an extension of .LST. *TK3*, though, uses a common file name that is used for all list files (TK3ASM.LST).

Click on the LIST button in *TK3*'s Assembly zone to open the LST file created when TK3TUT2.ASM was assembled, and print it to paper. Then assemble TK3TUT3.ASM and print out its LST file. A section of it is repeated in Listing 3A.

Examining both printouts you will see that the Program Count and Code Value hex numbers are the same in both listings, except for the last two (new) commands of TK3TUT3.

The LST files also include the programmer's notes at the right. They have been omitted from Listing 3A to conserve space.

The lefthand column (List count deci) holds the text line numbers as encountered in the text file listing (ASM) and are in decimal. They serve no programming purpose and are simply there for your information.

The second and third columns show the actual address location number (in decimal and hex respectively) within the PIC at which the command will be placed.

Command ORG 0 causes its associated GOTO 5 statement to be coded at location 0, similarly with ORG 4 and its GOTO 5 statement for location 4.

Note how columns four and five, which hold the 2-byte hex code and the equivalent binary (14-bit) value (not shown in Listing 3A), are only used if a command is encountered. For example, command GOTO 5 generates the code 28 05 (H'2805'), and command CLRF PORTB generates the code 01 86 (H'0186').

The .HEX file holds the code values in hexadecimal. The 14-bit binary value is that which is converted from the hex value and sent to the PIC as serial data.

TK3TUT3 FOREVER!
Now load the code for TK3TUT3.HEX into your PIC. It will be seen to start off in the same way as TK3TUT2. Now, though, when it gets to the end of the code, the l.e.d.s will all go out and the sequence will repeat, indefinitely! You will find that a fully-clockwise setting of VR1 becomes preferable, to increase the display rate.

The program difference now is that there are two extra commands and when command BSF PORTB,7 has been performed, Port B is cleared and the program follows the command GOTO LOOPIT. LOOPIT is the name given to the address at which the command BSF PORTB,0 has been placed. During assembly that address number has been noted and each time the assembler encounters a command reference to the address named LOOPIT, it substitutes the number for that address. There is just one reference in this program, but other programs may have many such references.

Names, when given to program listing addresses, as with LOOPIT here, are commonly known as "labels". Referring to the .LST listing for TK3TUT3, the numbered line 0018 reads:

0018 9 0009 14 06 LOOPIT BSF PORTB,0

Note the value 9 in column 3. Now look at line 0027, which reads:

0027 18 0012 28 09 GOTO LOOPIT

Now note the 09 in column 5. The two values are equal and intentional. The address for which LOOPIT is the reference name (label) is at location 9; the code 28 09 contains the instruction to GOTO (jump to), plus the address number to which it is to jump.

In other instances, the addresses may be much greater than the one illustrated here, and the two values will differ accordingly, but the point is that the name LOOPIT is replaced by an address value during assembly and as such is treated by the Assembler in the same way as were STATUS and PORTB.

This fact has another important significance: when a number has a name

allocated to it, each time the assembly program encounters that name it substitutes the appropriate number. Names can be given to addresses (as just illustrated), to register files (as with PORTB), and even bit numbers (Z to represent the Zero flag bit of the STATUS register, as shown later).

We shall also show examples of names being used as pointers to addresses when the address required may depend on a particular value established as a result of calculation, i.e. in the case of Indirect Addressing, which will be examined in Tutorial 16.

It is worth noting that *TK3* allows labels to be suffixed by a colon (:), e.g. LOOP:, which makes labelled routines easier to find using a text editor's Search or Find facility in a long program that has many calls to a particular label. Microchip's assembly programs do not permit this.

TABLE 5. SPECIAL FUNCTION REGISTERS

Register	Address	Bank	Tutorial
EEADR	09	0	25
EECOCN1	08	1	25
EECON2	09	1	25
EEDATA	08	0	25
FSR	04	0·1	16
INDF	00	0·1	16
INTCON	0B	0·1	18
OPTION	01	1	18
PCL	02	0·1	4
PCLATH	0A	0·1	14
PORTA	05	0	4
PORTB	06	0	4
STATUS	03	0·1	2
TMR0	01	0	18
TRISA	05	1	4
TRISB	06	1	4

EXERCISE 3

3.1. Do the same sort of program modifications that you did with Exercise 2, examining the results achieved. Also try changing the position of the LOOPIT address in the lefthand column, putting it alongside BSF PORTB,2 for example.

3.2. The command GOTO LOOPIT can also be put elsewhere; try putting it between BSF PORTB,4 and BSF PORTB,5 and see what the result is.

There is an easy way of moving it in this instance without actually doing so: put a semicolon (;) in front of the three lines following BSF PORTB,4. The Assembler will then treat these lines as comments and ignore them. The use of a semicolon is a handy way to temporarily omit commands when debugging programs (locating errors).

3.3. What happens if a semicolon is put in front of any of the three CLRF PORTB commands?

TUTORIAL 4
CONCEPTS EXAMINED

Naming numbers
Bit naming
Bit codes C, F, W
Bit testing
Carry flag
Conditional loop
Instructions BANK0 and BANK1
Instruction #DEFINE
Pin protection

LISTING 4 – PROGRAM TK3TUT4

```
; TK3TUT4.ASM
; using aliases, bit names and conditional loops

#DEFINE BANK0 BCF STATUS,5
#DEFINE BANK1 BSF STATUS,5

STATUS      EQU 3           ; STATUS register
TRISA       EQU 5           ; Port A direction register
PORTA       EQU 5           ; Port A data register
TRISB       EQU 6           ; Port B direction register
PORTB       EQU 6           ; Port B data register

W           EQU 0           ; Working register flag
F           EQU 1           ; File register flag
C           EQU 0           ; Carry flag

            ORG 0           ; Reset Vector address
            GOTO 5          ; go to PIC address location 5
            ORG 4           ; Interrupt Vector address
            GOTO 5          ; go to PIC address location 5
            ORG 5           ; Start of Program Memory

            CLRF PORTA      ; clear Port A data register
            CLRF PORTB      ; clear Port B data register
            BANK1           ; set for BANK1
            CLRF TRISA      ; set all Port A as output (clear direction reg)
            CLRF TRISB      ; set all Port B as output (clear direction reg)
            BANK0           ; set for BANK0

LOOP1       MOVLW 1         ; load value of 1 into Working register
            MOVWF PORTB     ; load this value as data into Port B
            BCF STATUS,C    ; clear Carry flag

LOOP2       RLF PORTB,F     ; rotate value of PORTB left by 1 logical place
            BTFSS STATUS,C  ; check if the Carry flag (bit 0) of the STATUS
            GOTO LOOP2      ; command is actioned only if PORTB is not yet 0
                            ; the program jumping back to address LOOP2
            GOTO LOOP1      ; command is actioned only when PORTB now = 0
            END
```

Command MOVLW
Command MOVWF
Command RLF
Command RRF
Command BTFSS
Command BTFSC
Command PCL
Program counter
Register PORTA
Register PORTB
Register TRISA
Register TRISB
Register PCL
STATUS register bit 0

CONNECTIONS NEEDED

All Port B to all l.e.d.s.
Capacitor C7 as 1μF
Preset VR1 set to minimum resistance (fully clockwise)

INSTRUCTIONS #DEFINE AND BANK

One concept that you are likely to see in PIC software is that of defining a frequently used command format as a single name. Each time the Assembler encounters that name during assembly, the defined command will be substituted in the coding. Two such definitions appear in Listing 4:

#DEFINE BANK0 BCF STATUS,5
#DEFINE BANK1 BSF STATUS,5

The command BANK0 is then used each time the programmer would otherwise key in BCF STATUS,5. Likewise with BANK1. It is not only shorter, but conveys another concept more clearly than would direct manipulation of STATUS bit 5, that of Banks (or Pages), which were referred to in passing earlier. We have shown that STATUS bit 5 switches between addresses H'00' to H'4F' and H'80' to H'8B'. In fact, the latter extends to H'CF', but addresses greater than H'8B' are not available to the programmer. Writing to them simply wraps them back to an address H'80' bytes earlier.

We have so far equated two file names and address values, STATUS as 5 and PORTB as 6. You will have seen that they actually represent three functions, the STATUS function which is accessed jointly at locations H'03' and H'83', and two functions for PORTB accessed at H'06' and H'86'. With the PIC16F84 all Special File Registers are held between H'00' to H'0B' and H'80' to H'8B' (see Tables 2 and 5).

It makes for an easy shorthand way of defining which group is which by giving them names. As Microchip refer to these groups as being in Bank 0 and Bank 1, these are convenient name types to use. This, then, is why the terms BANK0 and BANK1 have been defined as above: it is simply an easy to remember convenience. BANK0 holds the H'00' to H'0B' group, and BANK1 holds the H'80' to H'8B' group. Note that you may sometimes come across the term Page instead of Bank to represent the same concept.

DOUBLE EQUATING

In Listing 4, following the three definitions we see STATUS, PORTA and PORTB being nominated (EQUated) as in Listing 3, representing register addresses 3, 5 and 6 respectively. The names TRISA and TRISB have crept in, though, and they also relate to register addresses 5 and 6 respectively. Why two names for the same number? It is done for the convenient reason that we know address 6, for example, relates to registers which appear in both BANK0 and BANK1, but which have different functions, Port B's data and direction registers, respectively.

It saves confusion, therefore, to have a different name for each, even though their address numerals are the same. The name TRISB is given to Port B's direction register since this is the name given to that function in PIC datasheets. The name PORTB now simply refers to Port B's data register. Exactly the same convention is applied to Port A, using PORTA and TRISA as the names in relation to location 5.

Incidentally, as mentioned earlier, there is a command TRIS available as part of the command set of some early PICs. Microchip recommend that it should not be used since it has been deleted from the PIC16F84 and later chips. The same applies to the command OPTION. Neither of these commands will be discussed here. You will see the use of TRISA, TRISB and OPTION_REG in this *Tutorial*, but the terms are used as Register file names, not as commands.

Where Special Function registers have had their functions equated to a name that is similar, henceforth the new name will be used. For example, Port B will be referred to as PORTB, Port A as PORTA and Status as STATUS. Additionally, in order to avoid repetition of comments made in earlier listings, from now on listing comments will not always be shown here for situations that have previously been discussed. The full listings, however, show comments where appropriate.

PIN PROTECTION

It was said earlier that unused PIC pins should never be left as "floating" inputs. The easiest way to ensure that they are not is to set them as outputs and to set their output value to 0. This is why PORTA and TRISA conditions have been specified, even though PORTA is not actually used in program TK3TUT4.

BIT NAMES

All the numerals to which names have been allocated so far have been related to file (register) byte addresses. It is equally possible to allocate names to particular bits in a file byte. This is especially useful when individual bits of particular files perform specific functions. Three examples are shown in Listing 4:

```
C       EQU 0
W       EQU 0
F       EQU 1
```

BIT NAMES F AND W

We have already said that data can be routed either to files or retained in the Working register. A single bit code, either 0 or 1, determines which destination. This bit value statement is required following the comma used with some commands.

For example, take the two similar commands RLF PORTB,0 and RLF PORTB,1, the command RLF (which is discussed in a moment) tells the PIC that the value within the file then stated (in this case the file is PORTB) is to be rotated left (multiplied by two). The result of this rotation can either be put back into PORTB, using the 1 suffix, or held in the Working register for further use, using the 0 suffix. If the Working register is chosen, the value in PORTB remains as it was.

Again for easy human understanding, it is more convenient to give a name to the different conditions than having to remember numbers. So the file destination 1 is called F for File, and the Working destination 0 is called W for Working. All very logical and clear! The two example commands thus become RLF PORTB,W and RLF PORTB,F.

CARRY FLAG C

One bit of STATUS (see Table 4), bit 0, is the bit which indicates whether a Carry or a Borrow has occurred during some commands. (It is, incidentally, common to refer to such bits as being "flags": the flag is then said to be set or cleared by any action which affects it.) The Carry flag is frequently required to be read in most programs and it is convenient to also give it a name, in this case C, hence the setting-up statement:

```
C       EQU 0
```

The bit can be manipulated or tested by commands such as BCF STATUS,C or BTFSS STATUS,C (discussion of BTFSS comes in a jiffy or two).

Before going any further with the contents of Listing 4, load its code into the PIC (TK3TUT4.HEX). What you will see is that the eight individual l.e.d.s on PORTB are being turned on at the same time the preceding one is turned off. The movement will appear to be going from right to left, from bit 0 to bit 7 (LD0 to LD7), and restarting at bit 0.

There are several ways of doing this (and many reasons why you should need to). Two programming techniques are discussed here, the one in Listing 4, and then a much shorter one later in Listing 5. The one in Listing 4 demonstrates the use of the commands MOVLW, MOVWF, RLF, BTFSS, and how two loops can be "nested" and made dependent upon each other.

COMMANDS RLF AND RRF

Many of you will be familiar with the electronic concept of shift register chips. Data can be loaded into the register either serially (bit entry) or in parallel (byte entry). The data can be shifted to the "left" or "right" in the chip, in response to a clock signal. The shifted data can then be made available either serially as bits, or in parallel as a byte. When data is shifted left and read as a byte (parallel output), each shift has the effect of multiplying the data by two. Shifting to the right divides it by two.

Take the 8-bit binary code 00000100 (decimal 4), for example. If this is shifted left by one place, the result is 00001000 (decimal 8). If the code had been shifted right by one place, the result would be 00000010 (decimal 2).

Most files within a PIC are capable of having their data shifted (rotated) to the left or to the right (although doing so on the Special Function Registers may sometimes produce unpredictable results). The two commands are RLF and RRF (Rotate Left File and Rotate Right File).

Both commands have to be followed by the file which is to have its data rotated, then a comma and then the destination, either F or W. For example: RLF PORTB,F or RLF PORTB,W. If the W destination is chosen, the original contents of the file remain intact (the result going into W); they are only changed if the F suffix is used, which causes the result to be placed back into the file, over-writing its previous value.

There are two problems associated with rotating a file's contents left or right. For the first, consider the situation when a file (for the sake of example, call it PORTB) contains a value such as 11010111 (decimal 215); there are many numbers that could illustrate the point about to be made. Suppose the rotate left command RLF PORTB,F is given, all bits are rotated left by one place. The value retained in PORTB becomes 10101110 (decimal 174) which is definitely not 2 × 215; the original lefthand bit has vanished from this 8-bit byte – a 9-bit byte would be needed to show the correct answer.

RIGHT AND CARRY

Alternatively, suppose the rotate right command RRF PORTB,F is given, all bits are rotated right by one place. The value retained in PORTB becomes 01101011 (decimal 107), which is definitely not 215/2; the original right-hand bit has vanished from this 8-bit byte.

In some cases, of course, the intention of rotating left or right may have nothing to do with multiplying a value by 2. It may be that we simply want to change the position of the bits for another purpose, such as changing the commands sent to the outside world to turn equipment on or off. In this case, the arithmetic accuracy of the rotate result would be immaterial.

The other problem (although it can be used beneficially) is that bits rotated out from either end of the byte are rotated into the Carry bit of STATUS. Simultaneously, the previous value held in the Carry bit is rotated into the byte at the other end.

Suppose that the Carry bit is initially zero. In the first RLF example above, the original value of 11010111 would be rotated left and the result would be correct as shown (10101110) because the 0 has come in to the right from the Carry bit. However, the last lefthand bit of the original value (which is a 1) would now be in the Carry bit.

Suppose then that another rotate left is made. The bits within PORTB would be rotated left but, at the same time, the Carry bit from the previous rotation would now be rotated into PORTB from the right. The value held in PORTB thus becomes 01011101 (decimal 93), and again the Carry bit now holds the 1 from PORTB bit 7. Therefore, the next rotation will result in an answer of 10111011 (decimal 187).

To avoid a set Carry bit (*which retains the status last acquired anywhere in the program*) being rotated automatically into a file byte from the other end, the Carry bit can be cleared by the command BCF

STATUS,C prior to each rotate command, unless, of course you want a set Carry bit rotated into a byte.

Referring again to the display you see on the l.e.d.s at the moment, controlled by TK3TUT4, the Carry bit clearing technique is being used immediately prior to the RLF command. We shall show what happens if the Carry is not cleared when TK3TUT5 is viewed later.

COMMAND MOVLW

In Listing 4 is the command MOVLW 1. The MOVLW command (MOVe Literal value into W) is the command which allows literal values (numbers) contained within the program itself to be moved (copied) into the Working register for further manipulation. The range of values is from 0 to 255, i.e. an 8-bit byte. Command MOVLW 1 instructs that the value of 1 is to be moved into W. Literal values may be expressed in decimal, hexadecimal or binary, e.g.:

MOVLW 73 (decimal)
MOVLW H'49' (hexadecimal)
MOVLW B'01001001' (binary)

Literals may also be the address values of other files whose names have been specified at the head of the program, or they may be the values assigned to be represented by other words or letters. The following are all legal commands:

MOVLW STATUS
MOVLW PORTB
MOVLW W
MOVLW LOOP1

Respectively, the commands would move into W the address value of STATUS (which we have specified as 3), the address value of PORTB (6), the value assigned to be represented by W (0), the address within the program at which the command line prefaced by label LOOP1 resides (a value known only to the program – unless you examine the LST file).

An important point about any of the Move commands, such as MOVLW, MOVWF and MOVF is that the original value (source value) itself remains where it is and is unchanged. The value is simply "copied" into the destination specified. Having moved a literal value into W it can then be immediately moved into a specified file destination, or it can be used as part of a further manipulation.

COMMAND MOVWF

Following the MOVLW 1 command in Listing 4 is the command MOVWF PORTB. Command MOVWF (MOVe W into File) simply copies the contents of the W register into the file specified, in this case PORTB. Apart from the destination statement, no commas or other statements are needed (or allowed) with this command. The MOVWF command is the only way in which full bytes of data can be copied from W into other destinations. As used in Listing 4, it is the value of 1 which is copied.

COMMAND BTFSS

Another command we are introducing in Listing 4 is BTFSS, Bit Test File Skip if Set. What BTFSS does is to examine the status of the file bit specified in the remainder of the command (bit C of STATUS in this case: BTFSS STATUS,C).

The word Set now becomes the important one. The PIC is being asked to test if the bit specified is Set (i.e. is it logic 1?). There can only be one of two answers, either "yes" or "no". In programming (and digital electronics too) if the answer is "yes", then the answer is said to be "true". If the answer is "no", then the answer is said to be "false" (not true).

Now we come to a situation which some find difficult to grasp until they understand "what" happens when the validity of the question has been established. It's simple, though, once the facts are known!

The convention is that if a situation is "true" then it can be represented by logic 1, Conversely, if the situation is "false" it can be represented by logic 0. Logic 1 and logic 0 are, of course, the two states in which a binary bit can be. Hold this idea in your mind for a moment and consider the next fact.

We have shown that programs are stored as instructions in consecutive memory bytes. It has also been shown that these bytes are numbered from zero upwards (Listing 3A). Microcontrollers such as PICs keep track of which program byte number is currently being processed, and there is a counter which holds this information – the Program Counter (PCL, as it is named for the PIC, Program Counter Low). Unless told otherwise, when one instruction has been performed, the program counter is automatically incremented (a value of 1 added to it) and the next consecutive command is performed.

CHANGE OF ADDRESS

The program address number held by the PCL can be changed, either when the instruction is one such as GOTO or CALL, or by the user telling it to add another literal value to itself. The next instruction performed is that at the address pointed to by the new value. It will be seen, then, that if the value of 0 is added to the PCL, the next instruction is simply the next one on. If, however, the value of 1 is added to the PCL, then the next consecutive instruction is bypassed (skipped) and the one beyond it is performed instead.

For example, if the program counter is at 52, then normally it will automatically add one to itself and the next instruction will be that at 53, and the one after that will be at 54, etc. If, somehow, we intervene and add 1 to the counter while it's still 52, the counter will become 53 but will still add its own value of 1 to itself, making 54. The program will thus jump straight from 52 to 54, omitting the instruction at 53. Should the value of 0 be added, then, of course, the program will go straight from 52 to 53.

Coming back to BTFSS, we know that the answer will be either 0 or 1. When the PIC performs the BTFSS command, the answer is automatically added to the PCL. Therefore, still assuming a PCL starting value of 52, if the answer is true (1), the PCL has 1 added to it and so the next instruction performed is that at 54, as above. If the answer is false (0), then zero is added to PCL and so the instruction at 53 is performed, again as above.

Look again at Listing 4 and the command BTFSS STATUS,C, i.e. we are checking if bit C of STATUS is set (true). If if is true that the bit is set, then the 1 of the truth

answer is added to PCL and so the command GOTO LOOP2 is bypassed and that which says GOTO LOOP1 is performed. If STATUS bit C is not set (false) then the program simply takes GOTO LOOP2 as the next command because the 0 of the false answer is added to PCL. OK so far?

COMMAND BTFSC

While this concept in still in your minds, let's look at the command which is the opposite of BTFSS, namely BTFSC (Bit Test File Skip if Clear). What this command does is to check if it is true that the bit being tested is clear (0). If it is true that the bit is clear, then the answer is 1. If it is false that the bit is clear (that the bit is not 0, but 1), then the answer is 0.

Let's see what happens in Listing 4 if we replace BTFSS by BTFSC. The command BTFSC STATUS,C tests the C bit to find out if it is true that it is clear. If it is true, 1 is added to PCL and command GOTO LOOP1 is performed. If it is false that bit C is clear, then 0 is added to PCL and so GOTO LOOP2 is performed.

We have, perhaps, somewhat laboured this explanation, but the concept of bit testing and the resulting action is one which causes some people problems, especially when testing for a bit being clear.

Why, they ask, is it that the answer is 1 if the tested bit is zero? Why does 1 equal 0? It doesn't, what you are looking for is the truthful answer to the question posed. Think about the question, think about the answer to it.

It is an important concept to grasp, and there are other situations where it occurs: when testing the Digit Carry and Zero flags of the STATUS register (bits 1 and 2, respectively). We shall encounter those situations in Tutorials 5 and 7.

LISTING 4 AGAIN

What you see the program of Listing 4 doing is the simple action of repeatedly "moving" an l.e.d. from right to left. There are only seven commands involved, yet, as witnessed by the length of discussion so far, there are several important commands and their concepts to be fully understood.

Let's relate those commands in simple terms to what is happening in the program.

First, at label LOOP1 the value of 1 is moved into W, this is then moved into PORTB, setting its bit 0 to 1 and clearing bits 1 to 7. As a result, the first l.e.d. at the right is turned on (LD0) and the others (LD1 to LD7) are turned off. In binary, PORTB's value is now 00000001.

Next, the Carry bit of STATUS is cleared to prevent it from interfering with the results of the rotate-left command that follows at label LOOP2 (as discussed earlier). You will see that this command is RLF PORTB,F. The F suffix means that the result of the rotation is retained in PORTB, and the contents of PORTB will have shifted so that the second l.e.d. (LD1) has come on because the 1 previously set by the MOVWF command has shifted from PORTB's bit 0 to its bit 1. Since the Carry bit was previously cleared, 0 is moved into PORTB bit 0, turning off l.e.d. LD0. The binary value has become 00000010.

Now the value of the Carry bit in STATUS is checked to see if a 1 has been shifted out from PORTB bit 7. In fact, it cannot have occurred yet since it takes eight shifts to bring the 1 from the right and

into Carry. However, the PIC is not aware of that fact, so the Carry bit has to be checked following each shift left.

If the Carry bit is not yet set, the command GOTO LOOP2 is performed, the program jumps back to that stated position and the RLF command is again actioned. As a result the third l.e.d. (LD2) will come on and the second l.e.d. (LD1) will go out, binary 00000100.

Eventually, after eight shifts, the 1 will have shifted through all eight bits of PORTB and into the Carry bit. At this point, there will be no bits set in PORTB, and so no l.e.d.s will be on. Now, on the test for the Carry bit being set, the answer will be true, command GOTO LOOP2 will be bypassed and the command GOTO LOOP1 will be performed, the program jumping to that label. The whole sequence then recommences by a 1 again being loaded into PORTB bit 0. As written, the program will repeat until the PIC is switched off or the Reset switch is used.

EXERCISE 4

4.1. What do you think will be the l.e.d. display sequence if another value is loaded into PORTB via the MOVLW command? Try any multiple of 2; then try any value that has more than one bit set, using the binary format, e.g. B'01001100'.

4.2. Also see what happens if the command RRF PORTB,F is used instead of RLF PORTB,F. What do you think will happen if you replace PORTB,F by PORTB,W? Then see what happens if BTFSC is used instead of BTFSS? (It is a common mistake to use the wrong command in this sort of situation.) Now swap the two commands GOTO LOOP2 and GOTO LOOP1.

4.3. Just out of interest, also try deleting the command BCF STATUS,C (just put a semicolon in front of it).

A SIMPLER ROTATION

Load TK3TUT5.HEX – it will be seen to be shifting the l.e.d. display to the left, as occurred when TK3TUT4.ASM was first run as TK3TUT4.HEX (before you started changing it – although, hopefully, you saved each variant under a different name).

You should notice that TK3TUT5 is running a bit faster than TK3TUT4 did. This is

because there are now fewer commands to process for the same result. Simplicity of code usually makes for faster processing speeds (or, rather, the fewer commands that need to be processed to perform a particular function, will result in a faster processing speed). Look at Listing 5 and you will see how few commands there are in the loop, just two. Let's examine the program flow.

In the full listing everything up to the statement BANK0 is the same as in TK3TUT4. Then advantage is taken of the fact that a set Carry bit will be shifted into a file when it is rotated left or right; the command BSF STATUS,C is given before the loop, so setting the Carry bit. Now when PORTB is rotated left with the command RLF PORTB,F, the Carry bit comes straight into PORTB bit 0, turning on l.e.d. LD0. Simultaneously, the Carry bit is cleared (remember why?).

The next command is GOTO LOOP, which the program does, again to rotate PORTB, causing LD1 to come on and LD0 to go out. For eight rotations left, the Carry bit remains clear, then on the ninth rotation the original 1 that has traversed PORTB will drop into the Carry bit, to be rotated back into PORTB on the next rotation. And so it goes on, indefinitely.

There are numerous situations in which rotation occurs and when the setting of the Carry bit is desirable. In this way, several files can be coupled as a very long shift register, e.g.:

BSF STATUS,C
RLF FILE1,F
RLF FILE2,F
etc. to
RLF FILE15,F

EXERCISE 4 CONTINUED

4.4. What happens if you add another RLF PORTB,F after the first? And if you add a third RLF PORTB,F?

4.5. What happens if you substitute a W for the F in one of the statements?

4.6. What happens if you amend the program to work with PORTA (changing the l.e.d. connections again) – why does the sequence not repeat with the Carry bit rotating back into PORTA? (What is different about PORTA and PORTB?)

TUTORIAL 5
CONCEPTS EXAMINED

STATUS bit 2
Zero flag
Bit code Z
Command MOVF

CONNECTIONS NEEDED
All Port B to all l.e.d.s.
Capacitor C7 as 1μF
Preset VR1 set to minimum resistance (fully clockwise)

It is appropriate at this moment to introduce a command allied to the Carry bit tests, testing the Zero flag bit of the STATUS register. This is bit 2 and in the heading of program TK3TUT6, shown in Listing 6, the letter Z has been equated to it:

Z   EQU 2

LISTING 6 – PROGRAM TK3TUT6

```
; TK3TUT6.ASM
; Using RRF and Z, Status bit 2
; Zero flag use, Command MOVF

        (Definitions, through to
        BANK0 as previously)
LOOP1   MOVLW B'10000000'
        MOVWF PORTB
        BCF STATUS,C
LOOP2   RRF PORTB,F
        MOVF PORTB,F
        BTFSS STATUS,Z
        GOTO LOOP2
        GOTO LOOP1
        END
```

The two opposite commands for zero testing are BTFSS STATUS,Z and BTFSC STATUS,Z, identical to the Carry checking commands except for the change of final letter.

We also take the opportunity to formally demonstrate command RRF (Rotate Right File). It was described in Tutorial 4, but not shown. You probably used it, though, when experimenting with Exercise 4. Thirdly, the command MOVF is introduced and demonstrated. Run TK3TUT6.HEX and refer to Listing 6.

The l.e.d. display controlled under TK3TUT6 should be seen to be rotating right, but otherwise the display repetition should be as seen in TK3TUT4 and TK3TUT5. The program opens up with the necessary initialisation commands. The command at LOOP1 is then seen to be MOVLW B'10000000' instead of the previous MOVLW 1. The set bit (1) is now at the left of the byte, instead of at the right (00000001).

This is moved into PORTB and the Carry bit is cleared, both commands as in Listing 4. At LOOP2, command RRF PORTB,F now replaces RLF PORTB,F, instructing the program to rotate to the right, the 1 moving progressively from bit 7 to bit 0 and then into the Carry bit. Next comes MOVF PORTB,F. Let's examine it.

COMMAND MOVF

Whereas MOVLW means moving a literal value into W, MOVF means MOVe File value. The file (PORTB in this case) is named following the command, but the command itself does not say where the value is to be moved (unlike MOVLW, where W as the destination is included in the command). The destination is stated by adding a comma after the file name and then adding either W or F, e.g.:

MOVF PORTB,W or
MOVF PORTB,F

Normally, the command would be used with W, so that the contents of the file are brought into W for presumed further use. At first, then, the concept of using F as the destination seems strange. Why move the file value back into the file without the value having undergone some sort of manipulation?

The reason is that many commands automatically affect various flags in the STATUS register (see Tables 1 and 4), setting or clearing them as appropriate. We have already

LISTING 5 – PROGRAM TK3TUT5

```
; TK3TUT5.ASM
; Showing how Carry bit rotates into
register
#DEFINE BANK0 BCF STATUS,5
#DEFINE BANK1 BSF STATUS,5

STATUS      EQU 3
TRISA       EQU 5
PORTA       EQU 5
TRISB       EQU 6
PORTB       DQU 6
W           EQU 0
F           EQU 1
C           EQU 0

            (ORG0 TO BANK0 as
            previously)

            BSF STATUS,C
LOOP        RLF PORTB,F
            GOTO LOOP
            END
```

seen the Carry flag being affected by RLF and RRF, but the Zero flag is not affected by these two commands, so a different technique has to be used to check for zero.

When command MOVF is performed, irrespective of the W or F destination, the Zero flag is affected. It is set if the file value is zero, cleared if the file value is greater than zero. So, if we wish to know whether or not the file value is zero, we can use the MOVF command to affect the zero flag, and do so without changing the file value. It is also significant that the contents of the W register are not affected when using the F destination and can therefore be used elsewhere if needed following the result of the zero check.

That is what is happening in Listing 6, moving F back into F to affect the Z flag, which is about to be tested in the next command, BTFSS STATUS,Z. What is being looked for is PORTB's value becoming zero after the 1 has exited from the right of the file (from bit 0).

The logic of BTFSS STATUS,Z is the same as that for the Carry flag. We are looking for the truthful answer to a question, in this case is it true (1) that the Zero flag is set (1)? The answer will only be true if the file value is zero (0) – another of those concepts which some people may find difficult to comprehend, a 1 being used to mean the presence of 0.

If the file value is greater than zero, i.e. does not equal 0, then the answer is false (0) and so the Zero flag is cleared (0). As with Carry testing, the result of the Zero test (1 or 0) is added to the program counter (PCL) and, depending on the result, either GOTO LOOP2 (Z = 0) or GOTO LOOP1 (Z = 1) is the command actioned. Consequently, LOOP2 commands will be cycled through eight times before a jump is again made to LOOP1.

EXERCISE 5

5.1. Prove that the Zero flag is affected by the command MOVF PORTB,W as well as MOVF PORTB,F (the proof is that the rotation is the same as before).

5.2. What happens if the B'10000000' of command MOVLW B'10000000' is replaced by another number? Experiment with different values.

TUTORIAL 6
CONCEPTS EXAMINED

Command INCF
Command DECF
Command INCFSZ
Command DECFSZ
Counting upwards (incrementing)
Counting downwards (decrementing)
Use of a file as a counter

CONNECTIONS NEEDED
All Port B to all l.e.d.s.
Capacitor C7 as 1μF
Preset VR1 set to minimum resistance (fully clockwise)

Load TK3TUT7.HEX and refer to Listing 7.

This first thing to notice in Listing 7 is that a new name, COUNT, has been added. It has been equated in the full listing as:

COUNT EQU H'20'

LISTING 7 – PROGRAM TK3TUT7

```
BEGIN       CLRF COUNT
LOOP1       MOVF COUNT,W
            MOVWF PORTB
            INCFSZ COUNT,F
            GOTO LOOP1
LOOP2       MOVF COUNT,W
            MOVWF PORTB
            DECFSZ COUNT,F
            GOTO LOOP2
            GOTO LOOP1
```

This represents the first example of the use of an "ordinary" file (as opposed to a Special Register File). Such files are used for temporarily storing data values while the program is being run. In the PIC16F84 the file addresses that can be used for this purpose are held in Bank 0 between H'0C' and H'2F'.

In this program we could have actually placed COUNT at location H'0C' instead of H'20', but have used the later address because that is the first location available in many other PIC families, such as the PIC16F87x and PIC16F62x.

There are now two new commands illustrated in TK3TUT7, INCFSZ (INCrement File Skip if Zero) and DECFSZ (DECrement File Skip if Zero). Two allied commands, INCF (INCrement File) and DECF (DECrement File), will also be examined. The ability to increment (add one to) a value, or decrement it (subtract one from) has wide benefits in programming. Two such instances are keeping track of events through the use of counters, and of changing the values of flag bits (when in bit 0).

COMMANDS INCF AND DECF

The concept of commands INCF and DECF are extremely easy to follow. The first simply adds 1 to a file value, the other simply deducts 1 from a file value. If the file value is 255 (11111111 binary) when INCF is called, the value rolls over to zero. If the file value is zero when DECF is called, the value rolls over to 255. Whenever the result of INCF or DECF is zero, the Zero flag is set, otherwise it is cleared. Testing of the Zero flag can be performed using BTFSS or BTFSC as discussed in Tutorial 5.

Taking PORTB again as the example file, the command formats are INCF PORTB,W or INCF PORTB,F, and DECF PORTB,W or DECF PORTB,F. As previously discussed, the result of either command with a W suffix is that the new value is held in W, the file itself remaining unchanged. Conversely, the F suffix returns the new value to the file stated. Both F and W suffixes affect the Zero flag response. (Table 1 shows the flags affected by any command.)

COMMANDS INCFSZ AND DECFSZ

There are two commands which, respectively, can replace the INCF and DECF commands and which automatically test the Zero flag, taking the appropriate route depending on the truth of the answer. These commands are INCFSZ and DECFSZ, as defined at the start of this section.

Using PORTB as the example file, the command formats are INCFSZ PORTB,W or INCFSZ PORTB,F and DECFSZ PORTB,W or DECFSZ PORTB,F. If the result of any of these commands is zero, the Zero flag is automatically set, otherwise the Zero flag is cleared. The status of the flag determines the program routing in the same way as if the flag had been tested using BTFSS STATUS,Z or BTFSC STATUS,Z.

COUNTING UP AND DOWN

Listing 7 illustrates two loops, one counting up, the other down, alternating between the two after each 256 steps. INCFSZ is used in the first, DECFSZ in the second. Before entering the loops, at the label BEGIN the counter (COUNT) is cleared. Then at LOOP1 the command MOVF COUNT,W is given, followed by MOVWF PORTB.

You should now recognise what the actions do: they cause the value of COUNT to be output to PORTB. Next, the command INCFSZ COUNT,F is given, adding 1 to the value of COUNT, simultaneously checking if it has reached zero. An answer of not-zero (Z = 0) causes command GOTO LOOP1 to be performed.

Eventually, when COUNT has rolled over to zero, after 256 increments, GOTO LOOP1 is skipped (bypassed) and LOOP2 is entered where the command MOVF COUNT,W is performed, followed by MOVWF PORTB. These two lines are repeats of those at the start of LOOP1. We shall see later how duplicated lines of code can be avoided by using the code once in a sub-routine, calling it from any other routine that we wish.

Next, DECFSZ COUNT,F is performed, decrementing COUNT from the entry value of zero. COUNT thus rolls back to 255. Simultaneously, the command checks if COUNT has reached zero. If it has not, GOTO LOOP2 is performed. When COUNT has decremented to zero, command GOTO LOOP1 is performed and the cycle restarts, and so on.

It will be spotted that the use of a separate counter is not actually required in this example. We could increment or decrement the value of PORTB directly, but we are using COUNT instead to illustrate the use of a separate file to store data. We might, for example, want to increment COUNT, and then go off and do some other processing using COUNT's value, outputting that answer to PORTB instead.

EXERCISE 6

6.1. If you were to use INCF and DECF instead of INCFSZ and DECFSZ, what would be the necessary changes to the program?

6.2. What extra commands would be needed to start each loop with a non-zero value, while still counting until zero occurs?

6.3. What would happen if you had erroneously used W instead of F in one or other of the INC/DEC statements?

NEXT MONTH

In Part Two we show how to use switches, generate sound, perform timing, use 7-segment l.e.d. and alphanumeric l.c.d. displays, and have more fun with our command performance!