The Eejit's Guide to the PIC Microcontroller

Based on an original document in 1996 by John Morton
Extensively revised and updated by S.J. Katzen for the PIC16F84 © 2003

Introduction

Ever buy a singing birthday card? The sobering thought is that there is more computing power in such a card than there was on the entire planet earth in 1948 — the year I was born. The tiny processors that pack such a punch are known as **Microcontroller Units (MCUs)**. In 1998[1] it was recorded that hidden in every home were about 100 of these MCUs (and related microprocessors) and about 20 more lurked in the average family car. Where are they? Well, besides birthday cards, they give the intelligence to your mobile phone; DVD/CD/MP3 and Video players; television, radio, camera (digital or otherwise); microwave; washing machine; PC; smart card and on and on.

What is a MCU? Well it all began with the digital computers[2] developed largely in response to the computational requirements of the second-world war — for example code breaking and the Manhattan atomic bomb project. These dinosaurs (very large with little memory and brain) were implemented using either thermionic tubes (valves) or electro-mechanical relay switches. By the 1960s, with the advent of magnetic core RAM memories, magnetic drum/disk backup stores, solid-state transistors and early integrated circuits, electronic computers were commercial realities, although still very large and expensive.

In the early 1970s, advances in silicon integrated circuits (ICs) were such that simple computer central processor units (CPUs) could be fabricated on a single IC. The first of these in 1972 was the Intel 4004, shown in Figure 1, which was implemented with 2300 transistors (compare with over 20 million in a Pentium 4) and ran at a clock speed of 108kHz. Intel called their new baby a "Micro-Programmable Computer on a Chip", but the term **Microprocessor Unit (MPU)** quickly became the accepted term.

To construct a complete computing unit a MPU needs external data and program memory, input and output interfaces (ports), address decoding and clock circuits. By the late 1970s, MPUs with around 70,000 transistors were being produced (such as the Intel 8086 — the direct ancestor to the Pentium range). Many applications, such as a smart card, do not need such enhanced computing power. Size and price are the main considerations. In situations such as this, an alternative use of the extra silicon is to integrate memory, input/output ports and auxiliary circuits, such as timers, on the one chip. The first manufacturer to go in this direction in a big way was Motorola, with a particular application to car electronics.

The device used to illustrate these notes is the Microchip PIC16F84, introduced in 1994. Although there are various families in the PIC range, the features introduced here are common to most of these devices; in particular to the PIC16FXXX mid-range family members. For example, the more extensive PIC16F628 and PIC16F877. Indeed, although specific details may vary, the majority of microcontrollers work in a similar manner.

[1] New Scientist, vol. 59, no. 2141, 4th July, 1998, pp. 139.
[2] There were digital mechanical and electromechanical computers designed and to some extent used from the 1800s onwards.

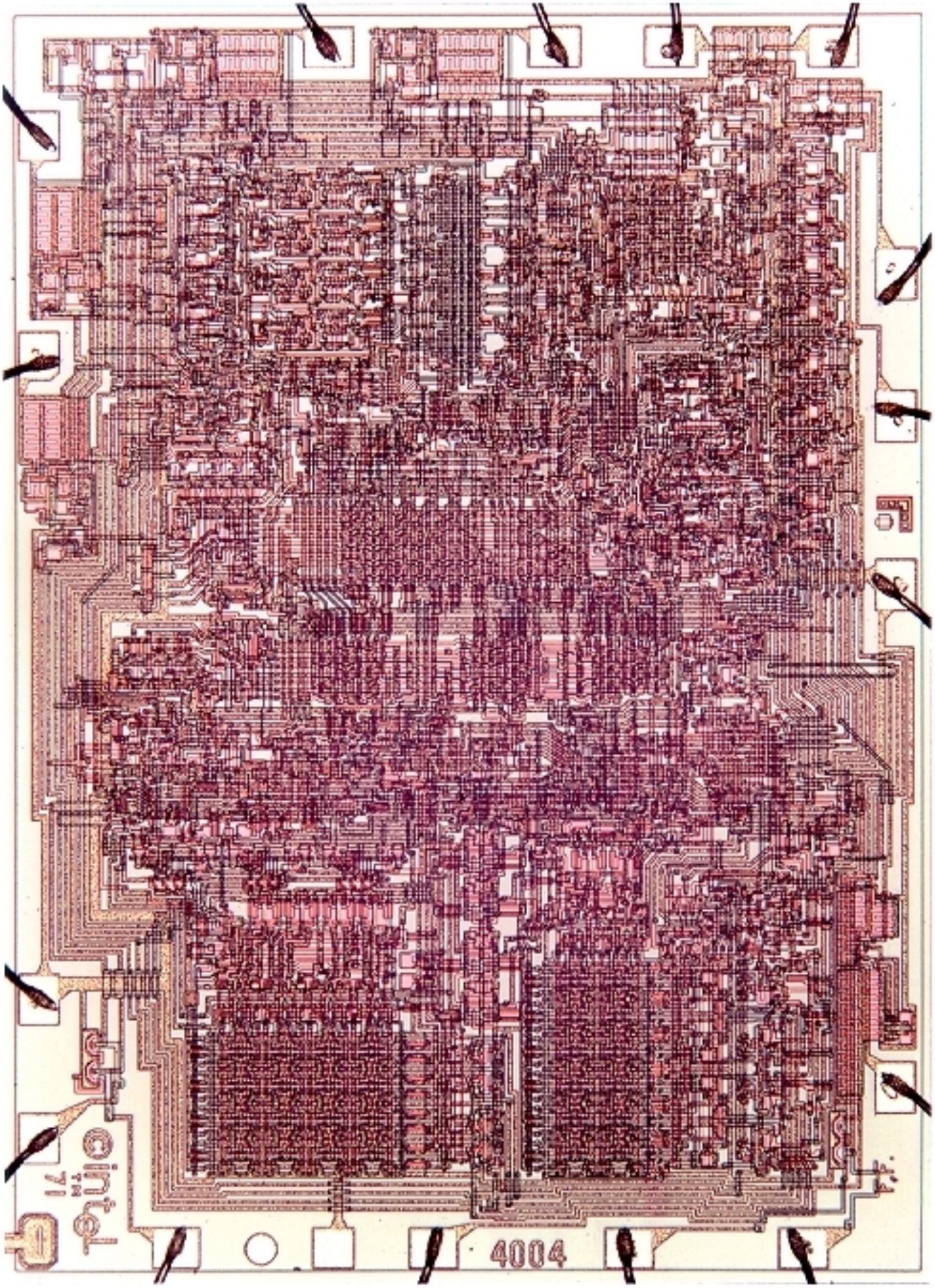


Figure 1: The Intel 4004 MPU; the granddaddy of them all.

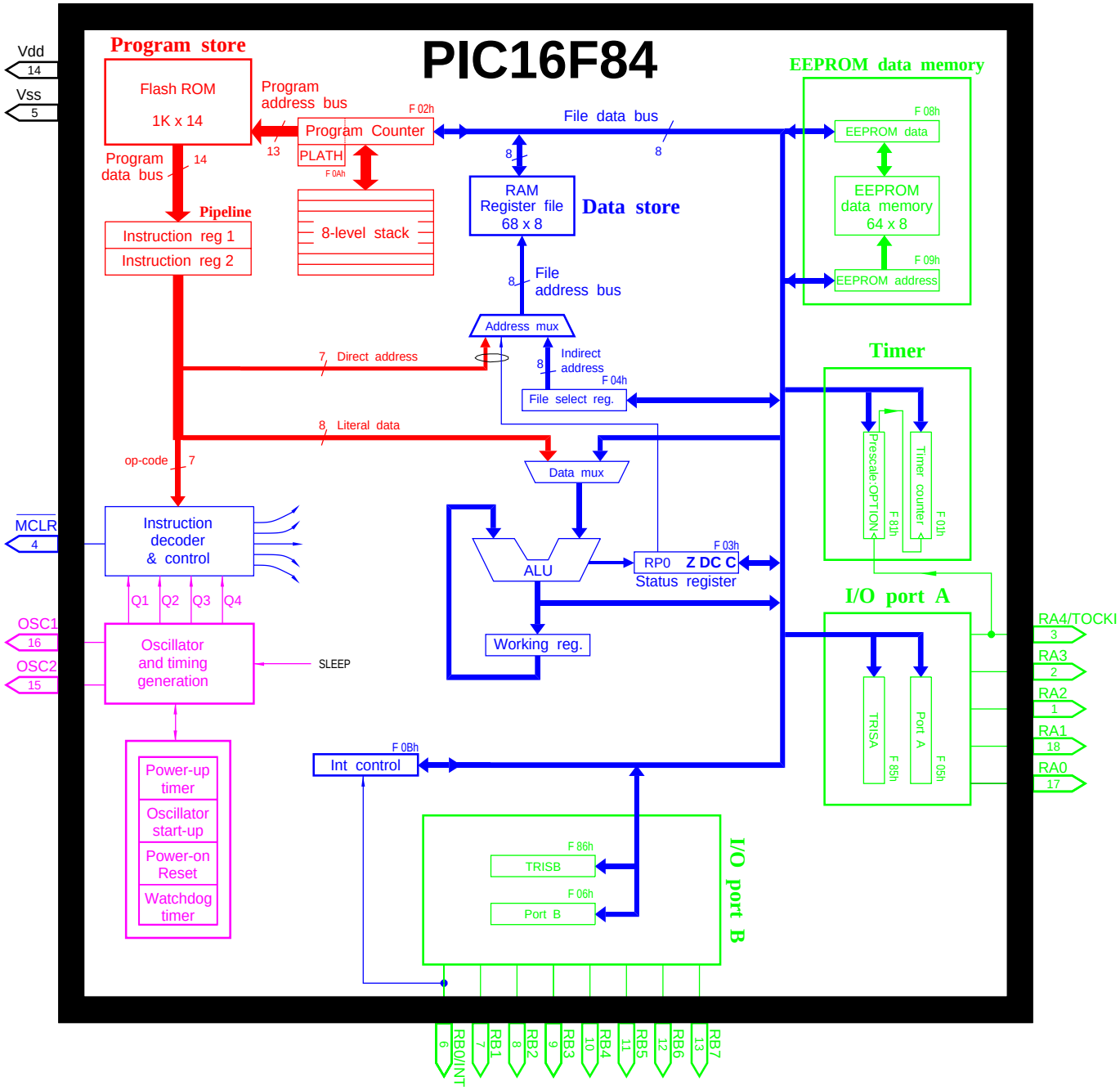


Figure 2: The PIC16F84 exemplar device.

When you buy a PIC microcontroller chip, you get a useless lump of silicon with amazing potential. It will do nothing without, but almost anything with, the *program* that you write. Under your guidance, almost any number or combination of normal logic chips as well as counting and multiplexing chips can be squeezed into one PIC MCU program and thus in turn, into one PIC MCU chip. PIC programming is all to do with manipulating numbers. There are only just over 30 instructions. Most of these are basic copy (move), jump, logic and arithmetic instructions. There are also instructions that allow you to clear, increment, decrement, set or clear individual bits in a datum. Conditional instructions allow the programmer to test any bit in any datum and to skip a following instruction depending on its state. This conditional skip may also be combined with an increment or decrement of a datum. In general, the chip simply processes these numbers as directed by the program to complete the designated task. These numbers can be:

- received from inputs, (by using '*ports*') from the outside world;
- stored in and loaded from compartments inside the chip (these are called *file registers*);
- added, subtracted, logic operations (e.g. AND) etc;
- or sent out through outputs, (by using 'ports') to the outside world.

This is, essentially it. There are certain functions inside the chip which make it easier to handle the numbers (for example Timer0 - a real time clock counter, which is explained further on, or '*flags*' which turn on in certain circumstances), but that is all that the PIC microcontroller is about.

The basic steps in programming chips are as follows:

- write program;
- assemble program;
- simulate / emulate program;
- program the chip;
- and finally test the system the PIC controls.

Hardware

The picture below shows how the pins on a PIC16F84 are arranged. The RA... and RB... types of pins are digital inputs and outputs (you can make them whatever you want). V_{DD} and V_{SS} are the positive and 0V supplies respectively, and the pin labelled T0CKI receives signals and counts them (if you program the chip to). T0CKI stands for Timer0 Clock Input. You may also see this labelled in older 12-bit PICs as RTCC — Real Time Clock Counter.

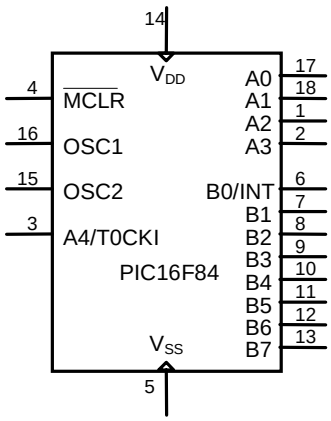


Figure 3: Pin-out of the PIC16F84

The $\overline{\text{MCLR}}$ pin is the Master CLeaR. The bar over the top means that it is active low. This means that when you make it low, the chip drops what it's doing and returns to start. In other words, a reset pin. Below are two diagrams showing how to implement a manual Reset circuit.

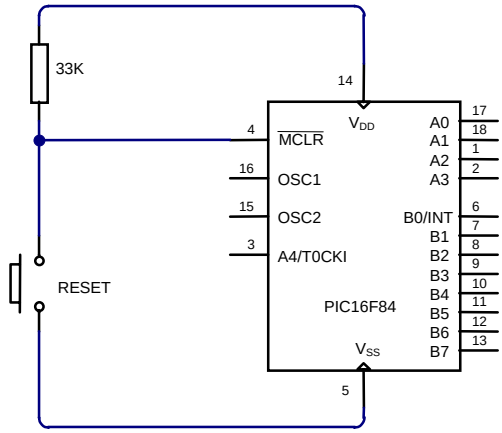


Figure 4: Resetting the PIC.

Figure 4 shows how to trigger the MCLR by means of a push button reset switch. The resistor is there because otherwise the positive and 0V supplies would be short-circuited when the button was pressed!

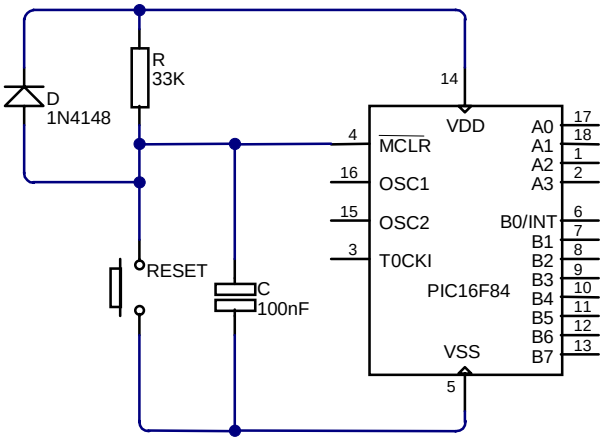


Figure 5: A better Reset circuit

In most cases, however, you will want to use the circuit shown in Figure 5 with the MCLR, since many power supplies take a short time to start up and crystal oscillators also need a 'warm up' before the circuit actually starts. This creates a small delay to allow for that. The diode discharges the capacitor quickly when the power supply is turned off.

The MCU needs some sort of steady pulse to keep it going; that is a **clock** signal. This can be done with a crystal or a RC (resistor - capacitor) arrangement. The most reliable is the crystal, as outside variables (such as temperature) don't effect it as much. Figure 6 shows how to connect a crystal to the internal oscillator. Typically values between 1MHz and 20MHz are used, but a 32.768kHz watch crystal is a popular choice where low power[3] is an important criterion and speed is not of the essence. Some PICs, such as the PIC16F278, have the option of a completely internal oscillator thus releasing valuable pins to be used as extra port input/outputs.

[3] The power dissipated is directly proportional to the frequency.

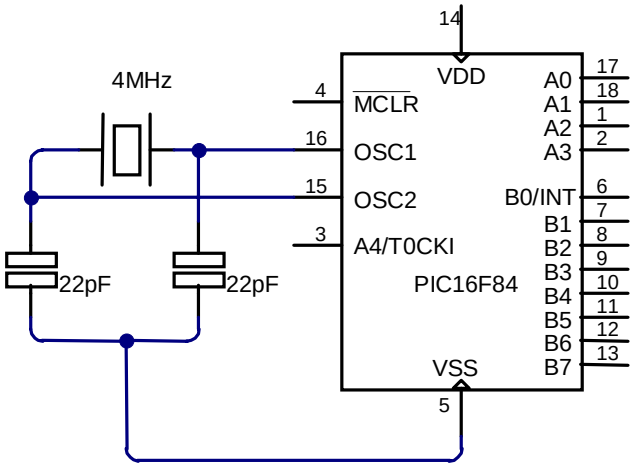


Figure 6: Clocking the PIC MCU (crystals can be up to 20MHz).

For more inputs and outputs there are larger PIC chips available. For example, the 40-pin PIC16F877 has 21 extra I/O pins, including additional peripheral modules, such as an 8-channel 10-bit analog to digital converter.

Now that you are familiar with the PIC MCU hardware, all you need now is know how to program. This booklet will hopefully give you the basics and as you program you should pick up little tricks and shortcuts along the way.

Memory

As explained in the introduction, programming is all to do with doing things to numbers. These numbers have to be stored somewhere. In essence there are two types of number.

1. Code representing data. In the PIC MCU each datum is represented as an 8-bit binary number. Another name for an 8-bit binary word is a *byte*. For example the number decimal 99 (represented in your software as `d'99'`) is actually stored as binary 01100011 (represented in your software as `b'01100011'`). An alternative representation to binary is hexadecimal, in this case 63*h* (represented in your software as 63h or `h'63'` or even `0x63`).

2. Code representing the executable instructions. Each instruction in the PIC MCU is stored as a 14-bit binary word.

More details are given in the following sections.

The Data store

Where are these data bytes stored? Unlike most microprocessors and microcomputers, the PIC family has a Data store (memory) completely separate to the memory used to store the instruction codes. In the PIC each data byte (datum) is stored in a *file register* and can hold *eight* binary bits. The organisation of this store is shown in Figure 7. You can think of the Data store as a filing cabinet with each folder holding *one data byte*. Actually there are two different types of folder in our cabinet. Some of these files are named and have special significance. These are known as Special-Purpose Registers (SPRs). The other unnamed files can be given names by the programmer and used for general-purpose storage These are known as General-Purpose Registers (GPRs). SPRs are used to control and monitor the state of the microcontroller. For example, File 3 is the Status register holding, amongst other things, the Carry flag (e.g. set when there is a carry-out after addition) and File 6 is Port B which is connected to pins RB7…0 shown in Figures 3 through 6 above. The PIC16F84 can hold 68 byte-sized general-purpose variables, addressed from File 0Ch through File 4Fh. Other PICs generally have more storage, although even the largest PICs have a maximum storage of 368 bytes. Even 368 bytes of RAM storage is not very much, so programs have to make very efficient use of this limited capacity! Think of your PC with hundreds of megabytes (a megabyte is 2^{20} = 1,048,576) of RAM!!!

Actually the PIC16F84 has two drawers in its filing cabinet, each drawer has a potential to hold 128 (2^7) files. The Programmer can move from one to the other drawers at will by setting or clearing bit 5 in the Status register. Each drawer is called a **bank**. This bit can be thought as the key to open the particular filing drawer. From Figure 7 you can see that this bit is called **RP0** (rather confusingly Register Page 0 rather than RB0 — I suppose to distinguish it from pin RB0; that is bit0 of PortB). Actually the PIC16F84 makes very little use of these banks. All the GPRs are imaged in both banks; for example, File 20h and File A0h are the same! Apart from the SPRs used for the EEPROM module, which we will not deal with in these notes, only the registers for setting up the port pins to either input or output — known as the TRIS registers — and the Option register, which is mainly used for configuring the Timer0 module, are unique to Bank1. However, this is unusual, and most PICs have four banks (with two RP bits in the Status register to select them) and many unique GPRs and SPRs in the various banks. Generally, Microchip put the most used SPRs in Bank0 and this is where the PIC starts on Reset.

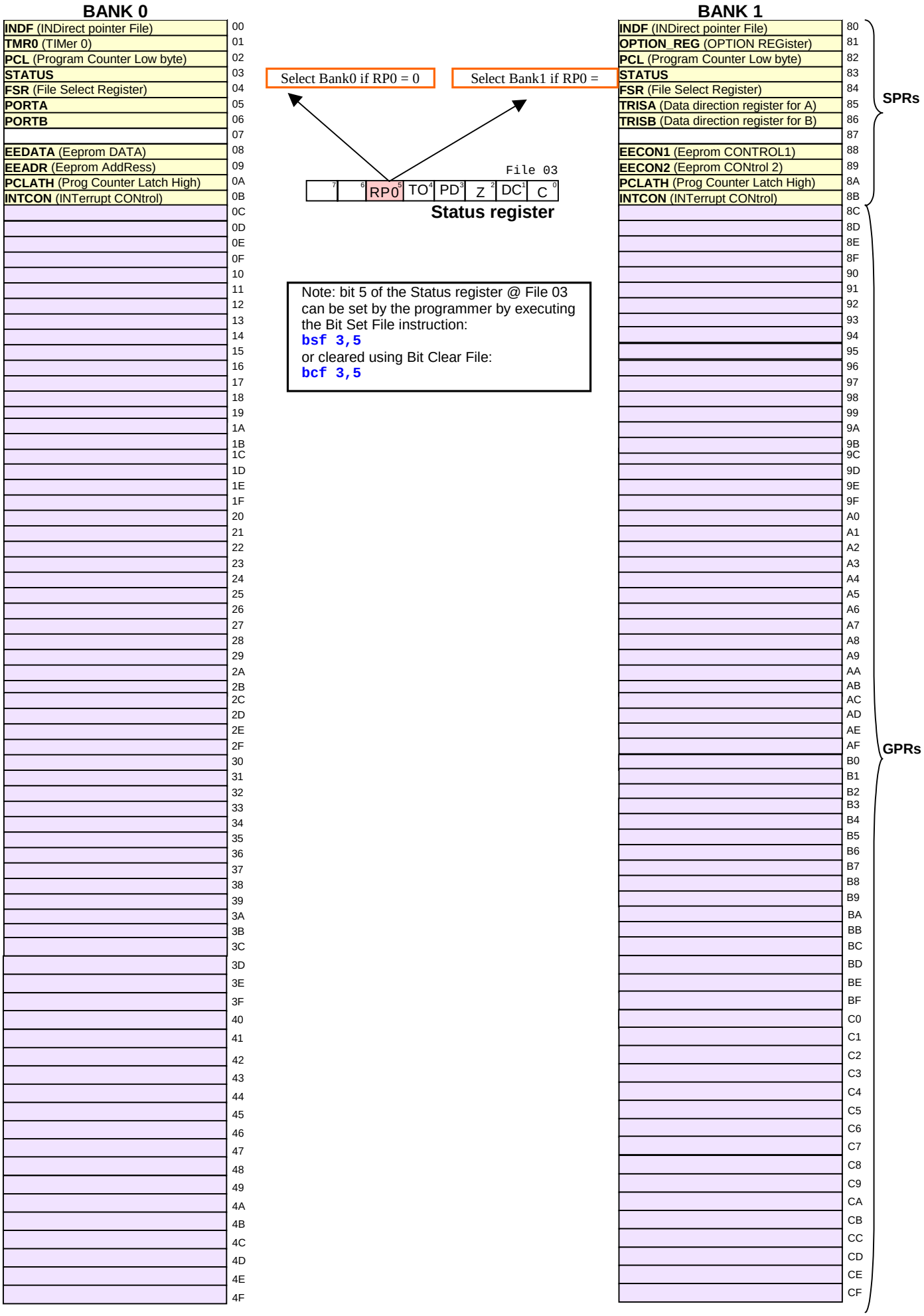


Figure 7 : The Data store of the PIC16F84

The Program store

Each instruction is held as a 14-bit word in the Program store. As can be seen in Figure 8 this store in the PIC16F84 has 1024 cells going from address 000*h* through 3FF*h*. Associated with the Program store is the **Program Counter** (called the Instruction Pointer in Intel microprocessors). On Reset the 13-bit Program Counter (PC) is cleared to zero — 000*h*. The PC thus addresses (points to) the first instruction in the program. This 14-bit instruction word is read into the first of a temporary register pair, called a **pipeline**. At the same time the PC is incremented to point to instruction 2. This is fetched into the top of the pipeline with instruction 1 going down to the bottom register whence it is decoded and executed. The process is continued with PC being incremented and instruction *n+1* being fetched from the Program store at the same time as instruction *n* being executed. This incremental progress continues unabated with each concurrent fetch/execute taking one instruction cycle (which is ¼ of the clock frequency). However, the Program counter's incrementation can be overwritten with a **`goto`**, **`call`** or **`return`** instruction, which causes the PC to jump to the address in question — e.g. **`goto 320h`** puts the 13-bit address 0 0011 0010 0000*b* into the PC causing it to jump to point to the instruction located in cell 320*h*. Also some instructions cause the execution to skip over the next instruction; i.e. the PC is incremented twice. In all such cases the pipeline has to be reloaded from the start, or *flushed*, and such instructions take *two* clock cycles to execute.

In the case of the PIC16F84 only the lower ten bits of the PC are actually used, hence the store size of 2^{10} = 1024 = 1K. With a full 13-bit PC the maximum store size is 2^{13} = 8192 = 8K; for example the PIC16F877. Thus no mid-range PIC can have a program with >8192 instructions.

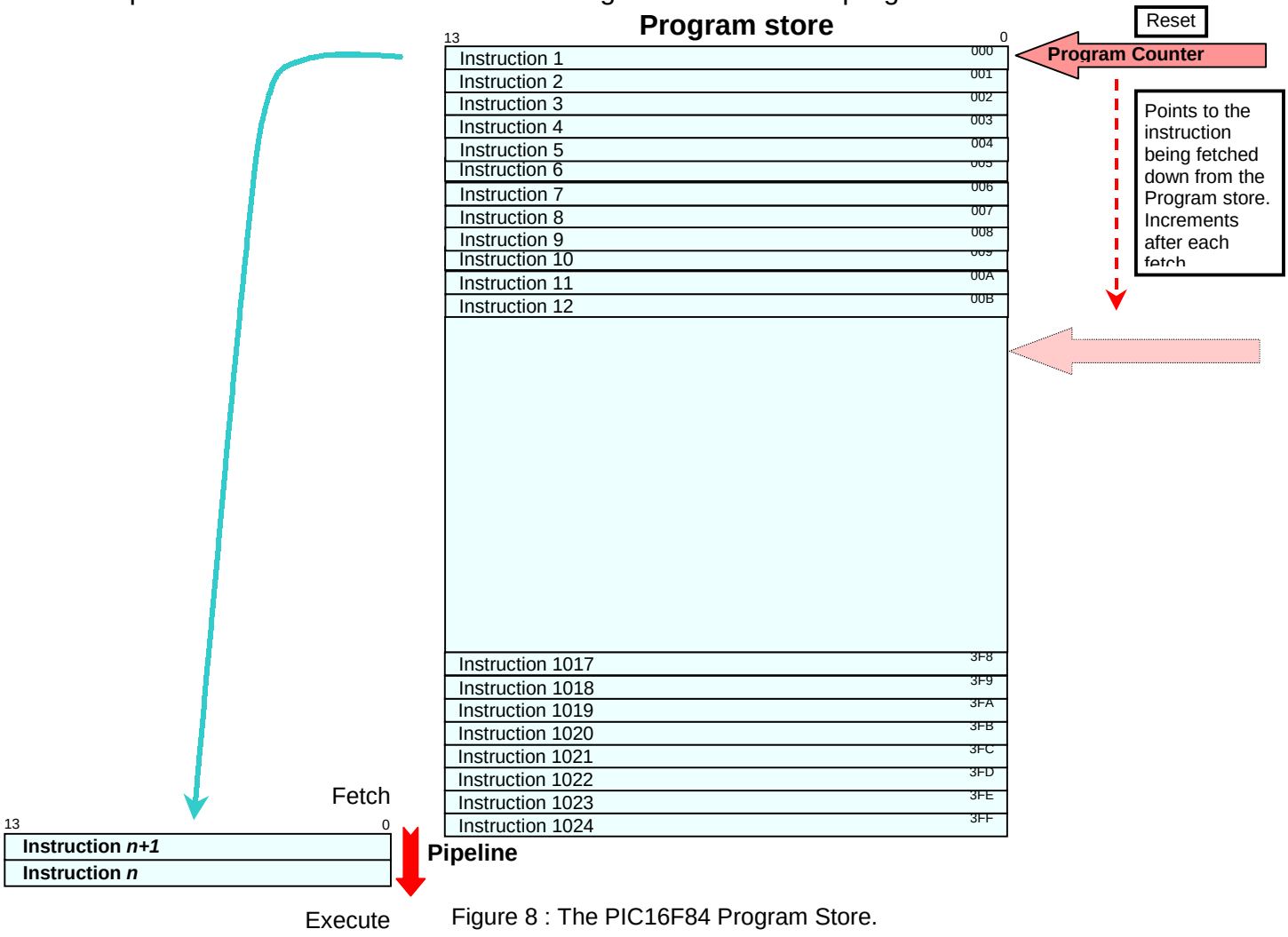


Figure 8 : The PIC16F84 Program Store.

Software

Simple Instructions

Our first example is to clear the contents of a specified file register. Remember, each file register has an address and can be thought of as a folder holding an 8-bit binary number between 00000000 and 11111111 (or if you prefer decimal 0 – 255 or hexadecimal 00 – FF):

```
clrf 20h  ; Clears File 20h (or clrf b'00100000')
```

This is your first line of programming and already it is easy to get confused. The instruction is **`clrf`** which tells the computer what to expect (i.e. what type of instruction is being executed). Then the number 20*h* is put after it, specifying which file register you want to be affected by the instruction. It is essential that after each line of programming you put some sort of explanation. Make sure you put a semi-colon before or the computer will interpret it as another instruction that it can't understand because even though a line may make sense as you write it, it might be incomprehensible when you come back to it later. A semicolon precedes all comments, and it is possible to have whole lines that are simply comments.

`clrf` stands for "CLear the File register" — in English this means "make the number in the file register zero". All the computer understands is binary numbers. Even instructions like **`clrf`** have to be changed into numbers for the program to be understood[4]. But, before you get worried, a computer-aided package called the *Assembler* does this for you. Some things that you write are interpreted immediately by the Assembler, but, there are other things (normally names which you have given certain things), which the Assembler can't understand. There is therefore a 'look up' file which defines the things you understand, into things the Assembler understands. An example of this is PORTA. Look at Figure 9. Ignore all the gobble-di-gloop except the pins with arrows going both ways. These are I/O (input/output) pins, which can each sink or give out up to 25mA. They are grouped into *Ports* that are divided into bits. RA0 means Port A, Bit 0, but to the computer it means something else altogether. A Port is addressed as a file; for example PORTA is in fact File 05 (see Figure 10), but it clarifies the program if we give names to things. For example, if we had wanted to clear (make low/zero) a particular bit in that file register we would write:

```
    bcf     PORTA,0   ; Clears RA0
```

which is rather more legible than:

```
    bcf     5,0       ; Clears RA0
```

It is a very good idea to put everything in the same format, this makes your program easier to read through and to debug (sort out any problems). Make use of the tabbing key in lining up instruction mnemonics, operands and comments. In all cases column 1 is reserved for labels (see page 12).

There is also an instruction **`clrw`** which clears the Working register.

[4] Actually the 14-bit machine code for clrf 20h is b'00000110100000'

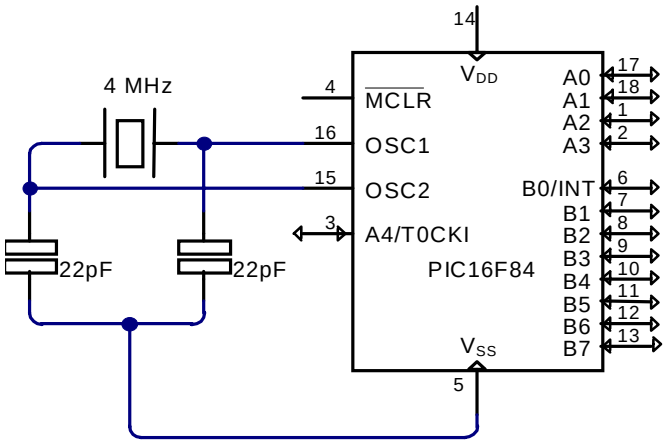
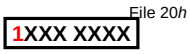


Figure 9: Showing input/output (I/O) lines

Some more instructions

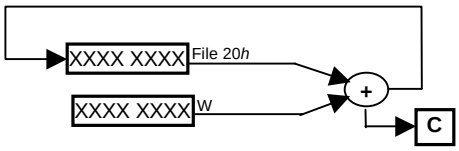
```
bsf   20h,7    ; Sets bit7 in File register 20h (makes it high/one)
```



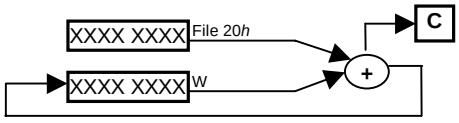

```
btfss 20h,5    ; Test bit5 of File 20h & skip next instruction if set

btfsc 20h,4    ; Test bit4 of File 20h & skip next instruction if clear

movf  20h,w    ; Copies the number in File 20h into the Working reg
```


```
movwf 20h      ; Moves (copies) the number in the Working reg to File 20h
```

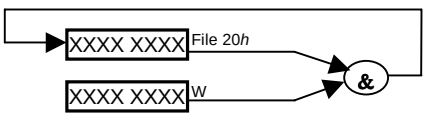

```
addwf 20h,f    ; Adds the number in the Working register with the one
in File 20h & puts the result in the file.  The ',f' after the file
register specifies the destination of whatever has been done.
```



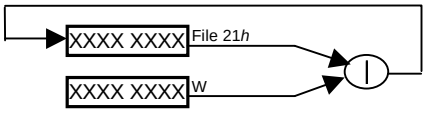

```
addwf 20h,w    ; As above except the result is put into the working
register. This is particularly useful when you don't want to change
the number in the file register, only use it.
```



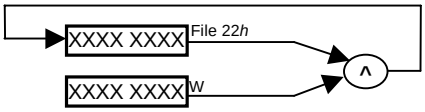

```
andwf 20h,f    ; AND each bit in W with each like bit in File 20h &
put outcome in File 20h
```



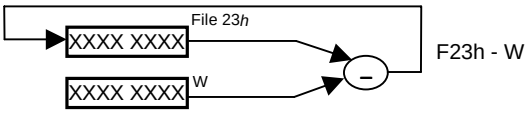
iorwf 21h,f ; Bitwise Inclusive-OR W with File 21h; outcome in File 21h



xorwf 22h,f ; Bitwise eXclusive-OR W with File 22h; outcome in File 22h



subwf 23h,f ; SUBtract W *from* contents of File 23h; outcome in File 23h



Most of these examples AND, inclusive OR, exclusive OR and subtract the 8-bit contents of the Working register with the contents of an 8-bit file register and put the result in the file register. To put the result in the Working register instead, replace **,f** with '**,w**' above.

goto ANYWHERE

this makes the program go to somewhere in the program which YOU have called 'ANYWHERE'. 'ANYWHERE' is a **label** in your program, a title for a certain section which does a particular job. The name should ideally be relevant to what that particular section does, but it is important not to give the same name to blocks of your program, as to file registers etc. I like to use uppercase letters to make labels stand out, but this is a matter of style. You can use up to 32 alpha-numerics plus underscore in making up names for labels. A label should *always* be in Column 1. An instruction *never* should be in Column 1!

Try to work out what this does! Don't be baffled, just go through it step by step and write in what instructions do on the lines by the instructions. Remember, PORTA is made up of inputs and outputs. You can make them whichever you want. Throughout the rest of the book, fill in the comments that you can!

There are also less general instructions used for specific things:

nop

this stands for No OPeration, in other words, do nothing, this may seem a pointless instruction, but in fact it will be useful, so remember it!

incf FileReg,f

increments (increases the number by one) a file register every time the program passes this point.

```
decf FileReg,f
```

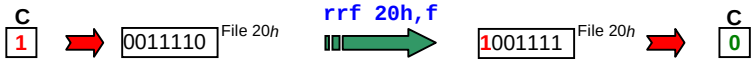
decrements (take a wild guess) a file register every time it passes this point. If you want the number in the File register plus one or minus one, but at the same time, not affect the contents of the File register, you must, like with **addwf**, and **iorwf** etc., replace **,f** by **.w**, and the augmented number will be stored in the Working register and will leave the original contents of the file alone.

```
incfsz    FileReg,f
decfsz    FileReg,f
```

will increment or decrement the contents of a file register, and if the result is zero it will skip over the next instruction.

```
rrf       FileReg,f
```

rotates the bits in a file register one step to the right; with the state of the Carry flag coming in from the left and the rightmost bit popping out into the Carry bit; for example:




```
rlf       FileReg,f
```

as above except to the left.

The **Carry flag** is used as an overflow bit in Rotate instructions. The normal function of this flag is to go high when something is carried over (a number grows larger than 255 or a bit gets 'knocked off the end' of a number). For Subtract instructions, the Carry flag doubles as a **NOT Borrow**; that is it is 0 whenever there is a borrow, and 1 if there is no borrow (sorry!).

Flags are kept in a Special-Purpose file Register (SPR) called **STATUS** at address File 03, as shown in **Figure 8** on page 10. The Carry flag is bit 0, so to do something to the Carry flag target this bit in File 03; for example to clear the Carry flag:

```
bcf  3,0  ; Clear bit 0 of the Status register
```

Remembering where the various bits are and the locations of the various SPRs is a bit of a bore and anyway using addresses and bit numbers makes your program difficult to read. The alternative is much clearer:

```
bcf  STATUS,C  ; Clear the carry flag
```

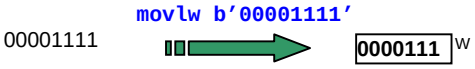
Now, with all these registers and bits and ports, it can be very confusing to think straight. Look again at the diagram in Figure 7 and see what you can or cannot recognise and what makes sense. This is the filing cabinet of the chip, many of the drawers are labelled (i.e. the Special-Purpose Registers (SPRs)) and some are blank (you can label then yourself). The first eleven (00 to 0B*h*) file registers for each bank each have a special purpose, and the rest are the general purpose ones. Some big chips have more files, as they have more built-in peripheral modules — e.g. the PIC16F877 has an 8-channel A/D converter, Serial port and more Parallel ports. However, they all have the basic set listed in the diagram — except some do not have the EEPROM module). Don't worry about unfamiliar names, most will be revealed afterwards.

Literal instructions

Most of the instructions above have dealt with moving or changing the contents of a File register. There are a few instructions that only apply to the Working register that copy or add a constant, or **literal** to W. For example:

```
movlw  kkkkkkkk ; Put the 8-bit constant number KKKKKKKK into W
```

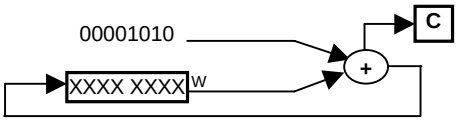
For example:



copies the constant 0Fh (or `movlw d'15'`) into the Working register.

```
addlw  KKKKKKK  ; Add the 8-bit constant number KKKKKKK to the
previous number in W
```

For example to add ten (0Ah) to the contents of W:



You can easily increment the contents of W:

```
addlw  1        ; Add one to the contents of W
```

and you can just as easily subtract one[5] by adding the 2's complement of one, which is FFh, but let the assembler do the 2's complementing for you!

```
addlw  -1       ; Adding FFH is the same as subtracting one!

addlw  -d'20'   ; Subtract twenty
```

Whatever you do, do not try and use numbers greater than decimal 255, hex FF or binary 11111111 with your literal instructions. An instruction addlw d'20000' in an attempt to add 20,000 to the contents of W has no meaning and your assembler will warn you of the problem. This is just as silly as trying to fill your 40-litre petrol tank with 200 litres across the border to save money; it will not work!

Special-Purpose File Registers

Indirect Addressing

First, we have the INDirect address File (INDF) located at File 00. This isn't a real file, as such, but an imaginary one, a gateway or *pointer to* any file register. When you specify the Indirect address File as a source or destination for a datum, it automatically uses the address in the FSR (File Select Register) located at File 04. You can think about it as a postman. You give him a letter and the address it has to be delivered to is to be found in the FSR. This means, for example, if you copy the contents of the Working register to the Indirect Address, and the number in the FSR happens to be 25*h*, then the contents of W will end up in File 25*h*! This can be useful when you systematically want to send or read data in an array of specific file registers.

[5] Although there is a `sublw` this actually subtracts W from the constant not the constant from W; for example, `sublw -1` generates 1 – W (which is the negative of what you are likely to want. As a general rule don't use `sublw` unless you know what you are doing!

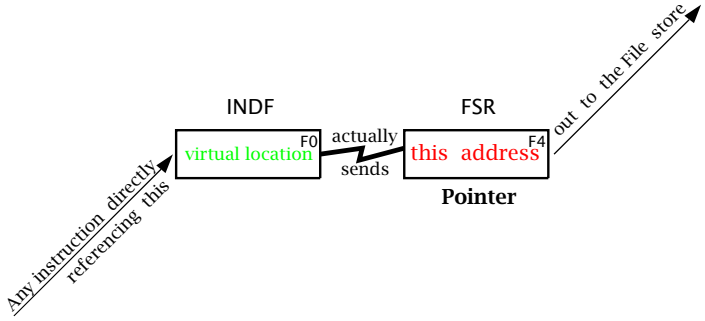
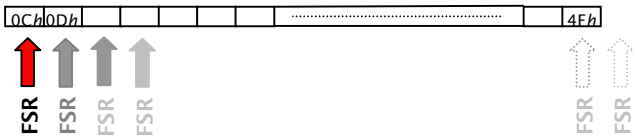


Figure 10: Indirect memory referencing using the FSR as a pointer.

An example of this, shown below, should be used at the start of your program to systematically clear all the general-purpose file registers on the chip. In the example below we start from File 0C*h*, the first of the general purpose registers, up to the last file File 4F*h*.

```
     movlw     0Ch                ; Initialise the FSR pointer by putting
     movwf     FSR                ; the literal 0Ch into it

LOOP clrf      INDF               ; Clear whatever the FSR points to
     incf      FSR,f              ; Increment the FSR

     movlw     50h                ; Now check if FSR has reached File 50h?
     subwf     FSR,w              ; by subtracting 50h from it (in W)
     btfss     STATUS,Z           ; and checking the Zero flag in STATUS
      goto     LOOP               ; IF not zero (i.e. equal) THEN do again
     ...       ...
```



The program loops starts with the FSR pointing to 0C*h*, and then clears the INDirect address File. This special file address of zero triggers the internal logic and it actually goes and looks at the address specified in the FSR, in order to find out where to send the number it has been given. In this case the Indirect Address has been sent a 00 (the equivalent of being cleared), but it actually ends up being sent to the first general-purpose register (file register 0C*h*). It then raises the number in the FSR and checks to see if it has gone one past the last file 4F*h*. This it does by subtracting 50h from the contents of the FSR (50h – [FSR]); putting the outcome in the Working register to avoid overwriting the FSR. We then check for zero (i.e. is 50h – [FSR] zero?) by testing the state of the **Z** flag in the Status register. If the Z flag is set showing Zero, then we skip out, otherwise we loop back again; thus systematically clearing everything from file register 0C*h* through File 4F*h*. This technique of using a pointer into memory is known as **Indirect addressing**.

The program above uses names for the Special-Purpose files **STATUS**, **INDF**, **FSR** and also the name **Z** for bit 2 in the Status register. How did the assembler program know that, say, **STATUS** was actually the same as the number 03? There are several ways to do this for SPRs. The first is to list them at the top of each program; for example:

```
INDF      equ  00   ; Tell assembler that the name INDF is EQUiv to 00
FSR       equ  04
STATUS    equ  03
Z         equ  2
```

Actually this is a bit of a bore; especially in the larger PICs with dozens and dozens of SPRs, so Microchip provide a header file for all its supported devices with all these **equ** directives listed. All the programmer has to do is to is include the appropriate file at the start of each program; for example:

```
include   "p16f84.inc"   ; Header file naming all SPRs and bits
```

and we have assumed that this line has been included in all our previous programs and for most of the others below.

Ports

File addresses 05 and 06 are special, for they are effectively connected to pins outside the chip, so can control or monitor the outside world. By that I mean a pin can be made high or low under the control of software or alternatively the state of a pin may be tested in software to see if it is high or low. The idea behind these **parallel Ports** is shown in **Figure 11** below. The term parallel is used as all pins connected to a port may be read or changed at the same time.

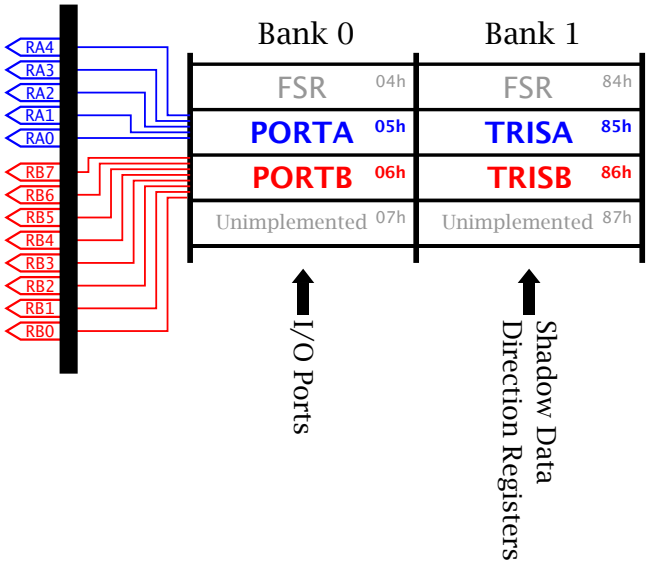


Figure 11: Simplified view of parallel ports.

Each bit in a file designated as a parallel port is effectively connected to an associated pin. Moreover, each pin may be set to be either an input or an output on a pin by pin basis. If a pin is set up to be an input (as are all parallel Port pins when the PIC is reset on power-up) then the associated bit in the file reflects the electrical state of the pin which is being driven from something in the outside world. Conversely if a pin is set up to be an output then the electrical state of the pin will follow the setting of the appropriate bit in the Port file. For example, if Pin RA4 is an output, then it will be close to 0V if bit 4 of File 05 is 0 and typically +5V if bit 4 is set at logic 1.

How do you set up the pins to be either inputs or outputs? Well, each port has a shadow file in Bank1 which is sensibly known in most microcontrollers as Data Direction registers. However, Microchip call these TRIS (TRI-State)[6]. Each bit in a TRIS register corresponds to the like bit in the PORT register; with a 0 for Output and 1 for Input — hence all TRIS

[6] The reason for this name is given in Chapter 11 of S.J. Katzen's *The Quintessential PIC Microcontroller*, Springer-Verlag, 2003.

registers are loaded with 1s on Reset. Thus, for example to make pins RB[7:5] and RA0 Outputs and the rest inputs we have:

```
bsf     STATUS,RP0      ; Change to Bank1 by setting the RP0 bit
movlw   b'11110'        ; Bit0 Output, bits 4-1 Inputs
movwf   TRISA           ; of Port A
movlw   b'00001111'     ; Bits 7-4 Outputs, bits 3-0 Inputs
movwf   TRISB           ; of Port B
bcf     STATUS,RP0      ; Change back to Bank0 by clearing RP0
```

As an example, assume that our PIC MCU has a 4MHz crystal and we want to generate a continuous square wave at pin RA0 with a frequency of 100kHz.

```
      include   "p16f84.inc"
      bsf       STATUS,RP0      ; Change to Bank0
      movlw     b'11110'        ; Make pin0 an Output
      movwf     TRISA           ; of Port A
      bcf       STATUS,RP0      ; Back to Bank1

LOOP  bsf       PORTA,0         ; Pin RA0 high    1~
      nop                       ; Do nothing for  1~
      nop                       ; and again       1~
      nop                       ; and again       1~
      nop                       ; and again       1~
      bcf       PORTA,0         ; Pin RA0 low     1~
      nop                       ; Do nothing for  1~
      nop                       ; and again       1~
      goto      LOOP            ; This takes      2~
```

Initialisation

Endless loop

Program 1: Creating a 100kHz square wave.

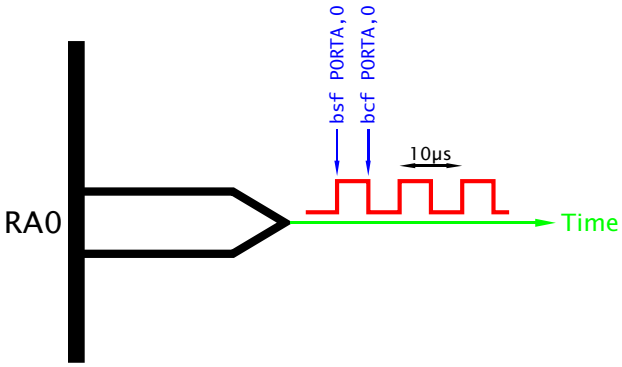


Figure 12: Generating a 100kHz square wave signal.

To get the timing right, we have to remember that the processor cycles through the instruction in cycles at ¼ of the clock oscillator frequency. In this case with our 4MHz crystal we have an instruction cycle frequency of 1MHz giving a duration of 1µs; thus we need to set the pin high for 5 cycles and low for 5 cycles. Some instructions take two cycles to execute as they need to flush the pipeline, as discussed on page 9. The **goto** instruction above is an example of such an instruction, as it disrupts the simple incrementation of the Program Counter.

Self Assessment Question: How could you alter the program if the crystal frequency was 8MHz?

Program Counter

File 02 is labelled **PCL** (for Program Counter Low byte) and File 0Ah is labelled **PCLATH** (Program Counter LATch High). PCL is actually the lower 8 bits of the Program Counter that points to the instruction being fetched from the Program store (see **Figure 8** on page 10). In some cases, typically in writing a look-up table routine (see Program 4 on page 25), you want to alter the state of the Program Counter maybe to skip past a number of instructions rather than just one. In this case the number can be added to PCL.

You have to remember that the full PC is actually 13-bits wide and when you add to the low byte PCL the PIC actually moves the lower 5 bits of PCLATH into the top half of the full PC. Thus the programmer can with a little bit of effort actually change all 13 bits at a time.

An additional use of PCLATH is with the **`goto`** and **`call`** instructions. Both of these instructions have only 11 out of the 14 bits available for an instruction to specify the destination address. For instance, the instruction **`goto FRED`**, where **`FRED`** is located at 6F0*h* in the Program store is actually coded as 101 110 1111 0000, where 101 is the code for **`goto`** and 110 1111 0000 is the address of the instruction labelled **`FRED`**. This is fine for small PICs, such as the PIC16F84 with only a 1K Program store. Any device with more than 2K, such as the 8K PIC16F877 is going to have difficulties going to far off places. Bits 3 and 4 of PCLATH may be used by the programmer to setup the missing bits 13 & 14 and allow jumps to anywhere in a 13-bit address space.

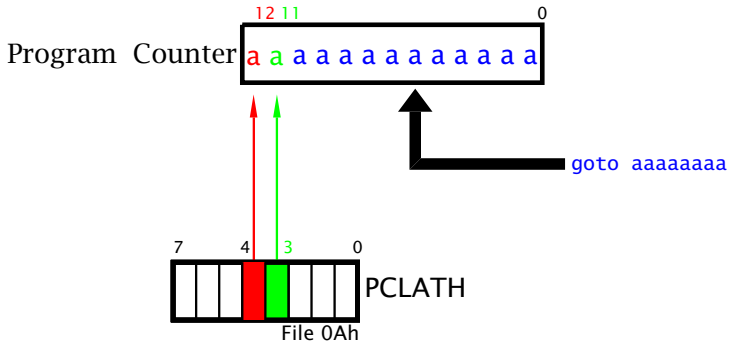


Figure 13: How the PCLATH register is used to create a 13-bit PC address.

Both PCL and PCLATH are zeroed on Reset.

Timer 0

Then we come to TMR0, or Timer Zero, which is located at File 01. TMR0 automatically counts either a signal derived from the crystal you connected to OSC1 and OSC2 or pulses from outside the device at the pin shared with RA4 — which in **Figure 3** on Page 5 is also labelled T0CKI (Timer0 ClocK In). You have to specify how many times you want it to count per signal and to do this you need to set up the Option register (OPTION_REG) which is located in File 81*h* in Bank1.

As you can see from the diagram the source of counting pulses is selected using bit 5 of OPTION_REG, called T0CS (Timer0 Clock Source). If this is a 0 then instruction cycles are counted (¼ of the crystal frequency) and if a 1 then from the outside world via pin RA4. Bit 4 is used to set which edge of this external pulse causes the incrementation, if T0Se (TMR0 Set Edge) is 0 then counting occurs on a rising edge and if 1, then the falling edge. The Prescaler (actually a binary counter) can be used to divide down the counting frequency by powers of two

from ÷2 to ÷256. For example if the three PreScale bits[0:2] in OPTION_REG are set to 100 then it will take 16 pulses for the Timer0 File register to increment once. Unfortunately the Prescaler is shared with the Watchdog timer, discussed later, and bit 3 (PSA is PreScale Assign) must be cleared to 0 is a pulse scale is to be used. If it set to 1 then it will be assigned to the Watchdog timer and this is a way to directly increment Timer0.

When Timer0 overflows; e.g. F9 → FA → FB → FC → FD → FE → FF → 00 → 01…. Then bit 2 in the INTerrupt CONtrol register is set. Thus you can let the timer get on with it and monitor the state of **T0IF** (**Timer0 Interrupt Flag**) to check for this overflow condition. Better still you can set up the interrupt system to cause an interrupt automatically when overflow occurs, but we are not covering interrupts in these notes.

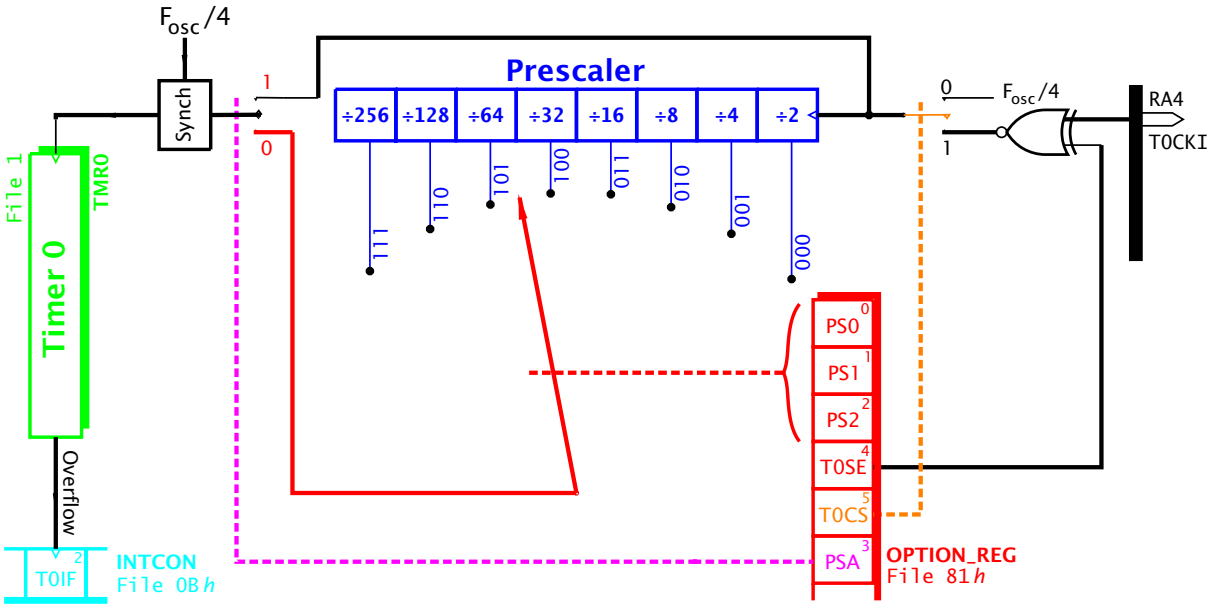


Figure 15: Timer 0.

To show how this all works, consider the following example.

Show how you would set up a PIC16F84 to count the number of people coming into a supermarket in batches of a hundred; that is increment a File register called, say, CENTURY, every hundred people coming past a sensor. Assume that the photocell is connected to pin RA4/T0CKI.

Program 2 below comprises five sections.

1. Initialisation of the Option register in Bank1 which is set up to select an external counting source and assign the Prescaler to the Timer to give a pulse frequency division ratio of ÷4 (I deliberately choose to increment the timer every four pulses and overflow on 25 of these to illustrate the use of the Prescaler). A better solution would be to assign the Prescaler to the Watchdog timer and count 100s directly. What changes would you make to do this?). You don't need to set pin RA4/T0CKI to input as this will be done automatically on Reset.

2. Back in Bank0 the batch count is zeroed.

3. Setting the timer file register to –25 (E7*h*) means that it will take 25 incrementations until overflow occurs. Remember that each increment occurs when four people break the beam.

4. The main part of the program simply monitors the state of the T0IF bit in the INTCON register. This is cleared on Reset and will be set only when TMR0 overflows.

5. When overflow occurs one is added to the batch-of-100 count and the T0IF flag is cleared. The process then continues from 3 above.

```
          include   "p16f84.inc"
CENTURY   equ       20h    ; File 20h used to count hundreds of people

MAIN      bsf       STATUS,RP0 ; Move to Bank1 to access OPTION_REG
          movlw     b'00100100'; T0CS External, Prescaler, ratio 4
          movwf     OPTION_REG ; Send to Option register
          bcf       STATUS,RP0 ; Back to Bank 0

; Now set up Timer0 to –25 (actually to E7h) so that it will overflow
after 25 pulses (25 times 4 to give 100).  Also zero the batch count
          clrf      CENTURY     ; Start batch count at zero

LOOP      movlw     -d'25'      ; Put –25 in Timer0 (i.e. File 01)
          movwf     TMR0
OVERFLOW  btfss     INTCON,T0IF ; Check out the TMR0 overflow flag
           goto     OVERFLOW    ; Keep checking

          incf      CENTURY,f   ; Its set! So add one to the Batch
count
          bcf       INTCON,T0IF ; Clear this flag
          goto      LOOP        ; and start all over again

          end
```

Program 2: Counting batches of 100 people.

Self Assessment Questions.

1. What is the maximum number of people this system can count and how could you increase this?
2. Can you think of any real-world problems that might arise in practice and how could you solve them?

Watchdog timer

Microcontrollers live in the real world with noise, power-supply fluctuations and even software bugs. If this disturbance should affect the processor then it will likely run berserk and end up executing incorrect code. In many cases manually resetting the processor will be all that is needed in the Microsoft way with your PC. However, in many situations a manual reset is not practicable and the consequences can be tragic; for example, if a pacemaker was microcontrolled or in a space probe in orbit around Mars!!!

To avoid this problem with a mission critical systems a designer can use a Watchdog timer which will automatically reset the device unless it is triggered on a regular basis as part of the legitimate program. All PIC MCUs have an integral Watchdog timer, which is shown below.

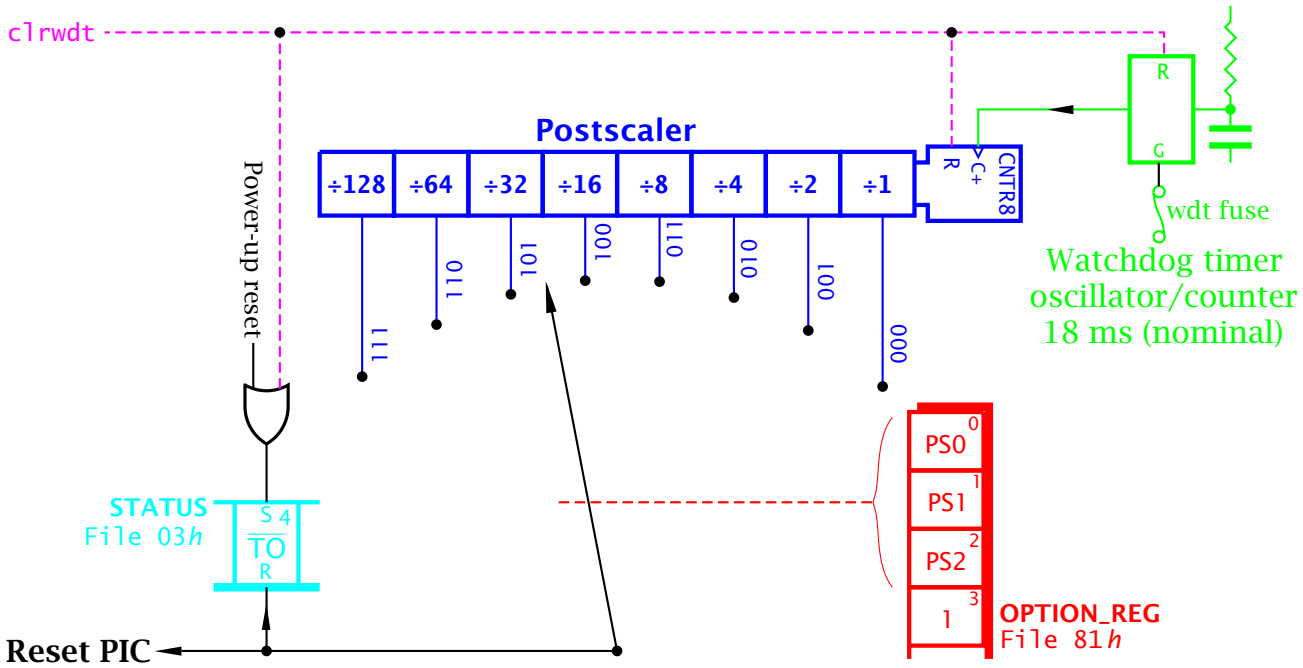


Figure 17: The integral Watchdog timer.

The Watchdog timer is in fact a simple monostable circuit which when triggered flips for around 18ms. This happens when the PIC is Reset and also when the instruction **`clrwdt`** is executed. Normally this would be part of the program inside an endless loop, so provided the instruction is executed no less than every 18ms the Watchdog will remain deactivated. If something goes wrong and the PIC goes berserk then this will not happen and the Watchdog will relax and reset the device.

From the diagram we see that the Prescaler we used for Timer0 can alternatively be used for the Watchdog timer to extend the basic 18ms period by up to ×128 (≈ 2.3 seconds), but remember that you cannot use this scaler for both functions.[7] If the Watchdog timer does timeout, as well as restarting the program at location 000*h*, it will also clear the **$\overline{\text{TO}}$** (Time Out) bit in the Status register. Thus the programmer can tell if the Reset was due to an internal Watchdog timeout.

Of course you may not want to use the Watchdog timer. If so you can disable it when you blast in your code. Rather than doing this manually, it is better to put a line in your program to configure the blasting software to set or clear the configuration fuses. Typically this may be something like this:

```
__config  _XT_OSC & _WDT_OFF & _PWRTE_ON & _CP_OFF
```

which not only turns off the Watchdog timer, but also configures the gain of the internal crystal oscillator, turns off Code Protection (to allow you to reprogram the device as many times as you like) and also enables the PoWeR Timer Enable which inserts a delay when the device turns on to allow the oscillator to stabilise.

Some PICs also have a Brown-out Detector which does something similar when the power supply drops below a fixed threshold.

[7] The PIC18XXX 16-bit series has separate scalers (and the option for a 16-bit Timer 0).

Subroutines

Even for something as simple as a PIC, programs can be extensive with over 8000 instructions. Bigger beasts can have several million lines of code. Just about always it is better to modularise your programs into short routines which have a well defined function that can be readily tested and documented. In this way testing and maintenance is easier and such modules can be reused for other programs and indeed the same code may be called from several parts of the one program. Not only can software modules be re-used between projects, but modules from other sources, like Microchip themselves, can be integrated into your own program. Thus, for example, you want a 32 × 32-bit multiplication? Just pull a module off the shelf from a Microchip manual or CD ROM!

All languages have provision for integrating software modules, but the name differs sometimes according to the language. The term **subroutine** is universal at assembly level (and also languages such as BASIC and FORTRAN). Other terms include Function and Procedure.

To use a subroutine there has to be some mechanism so that the processor remembers whence it was called, so on exit it can return to the caller at the correct point. For example, consider a subroutine that creates a 1ms (1000µs) delay. In the diagram the main program calls up the subroutine that begins at a location labelled DELAY from two different points. How does the processor know when it hits the **`return`** instruction the second time to return to a different point in the Program store? Well, the standard way is for the processor to automatically copy the value of the Program Counter at the jumping off point in the caller routine in a safe haven before overwriting it with the address of the first instruction of the subroutine. As the Program Counter is already pointing to the next instruction after the **`call`** instruction, when the **`return`** instruction is executed then this copy is pulled out from its hidey hole and put back into the Program Counter, thus effectively making execution move back to the instruction in the caller routine following the originating **`call`** instruction.

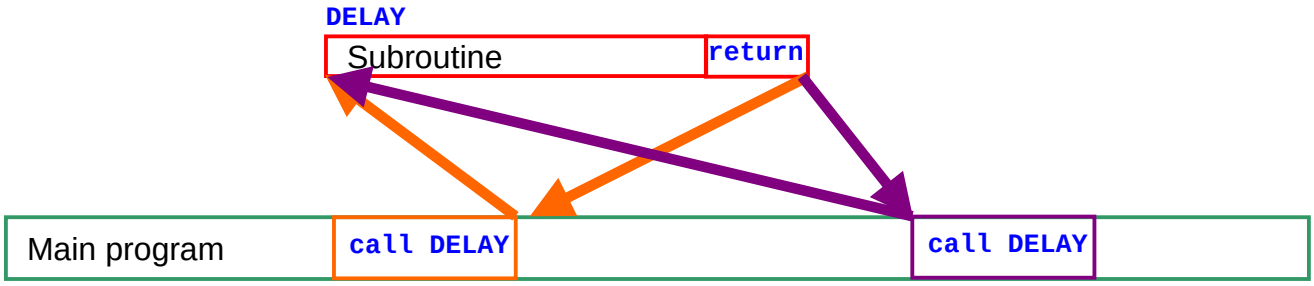
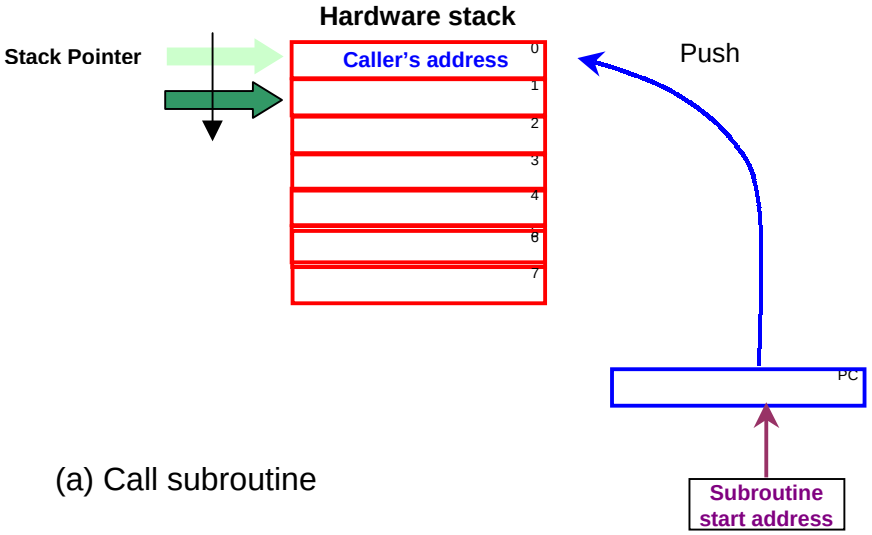
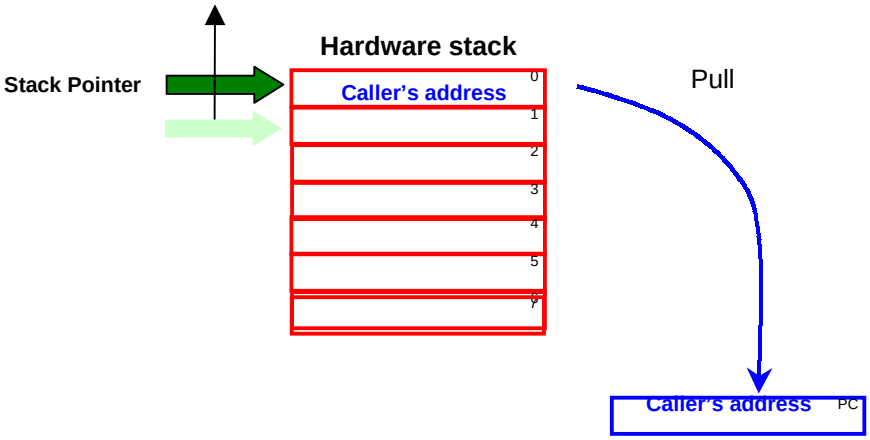


Figure 18: Calling up a subroutine.

The vast majority of processors use a special area set aside for the purpose of storing these return addresses. Normally this is an area of Data memory that the programmer has reserved for this purpose called a **stack**. In the PIC families a completely separate stack holding 8 return 13-bit copies of the PC is used for this purpose, as can be seen below.



(a) Call subroutine



(b) Returning

To activate this automatic save mechanism simple use the **`call`** instruction instead of the **`goto`** instruction. They are identical except **`call`** automatically pushes the jumping-off value of the PC on the stack before putting the new value in the PC, A subroutine should *always* use the return instruction to go back, *never, never* use a **`goto`**.

As there are eight locations in the stack, a subroutine can call another can call another up to 8 deep (seven if interrupts are used). As the path back from these **nested subroutines** is the opposite to the calling path the **last-in first-out** structure of the stack is ideal.

How do you write a subroutine module? Well there are no special instructions used on a subroutine apart from the rule that you must use one of the two return instructions:

```
return    ; Plain return to the caller program
retlw     ; Return as above, but with the specified constant in W
```

You should also put a label at the first instruction (entry point) which effectively names the subroutine and makes it easy to call.

Two examples will illustrate the use of subroutines

Example 1
Write a subroutine to create a 1ms delay, assuming that the crystal is 4MHz.

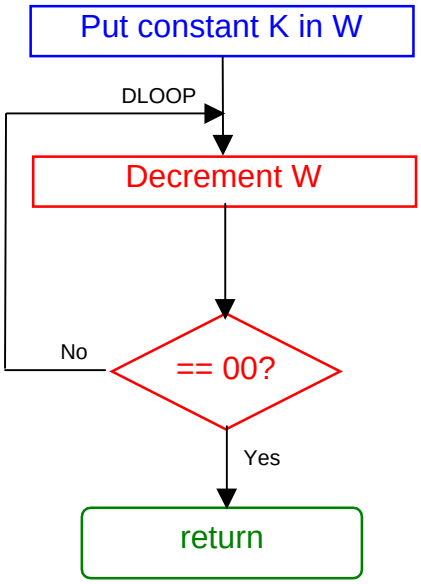


Figure 19: Flow chart for 1ms delay subroutine.

The flow chart for this subroutine is shown above. All we need to do is to put some constant, *K*, in the Working register and decrement it to zero. Remember, that the maximum value that can fit into W is 255, we must make sure that our execution time of 1000µs can be accomplished with a constant no larger than this. Also with a clock frequency of 4MHz, an instruction cycle will be 1µs — so we are looking for 1000 instruction cycles. Actually 998 cycles if we take into account the **`call`** instruction which we need to get to the subroutine which takes 2 cycles.

```
DELAY    movlw    K          ; We will work out the value of K later
DLOOP    addlw    -1         ; Adding –1 is the same as subtract 1!
         btfss    STATUS,Z   ; Did this set the Z flag?
          goto    DLOOP      ; IF not THEN do again
         return              ; ELSE return
```

Program 3: A 1ms delay subroutine.

And that is all there is to it! Now we need to calculate a value for *K*

`call DELAY`	2 cycles	From the caller program
`movlw K`	1 cycle	Puts the constant K into W
`addlw -1`	K × 1 cycles	Decrement done K times
`btfss STATUS,Z`	(K-1) ×1 + 2 cycles	2 cycles when skips out, else one cycle
`goto DLOOP`	(K-1) × 2 cycles	2 cycles each but skipped over the last loop
`return`	2 cycles	Return to caller

Grand total is 2 + 1 + *K* + (*K*+1) + (2*K*-2) + 2 = 4 + 4*K* = 1000 ∴*K* = 249

Thus we alter the first line to **`movlw d'249'`**.

Self Assessment Questions

1. How could you alter the subroutine if the crystal frequency was 8MHz?

2. What would be the delay if you loaded in zero into W?

3. Given that the maximum value that will fit into W is 255, how could you extend the basic core above to give a 100ms delay?

4. Based on 3 above, how could you extend the subroutine to give a $N \times$ 100ms delay, where N is a constant placed in W by the caller program?

Example 2

To extend Program 2 to give a 7-segment readout of the number of hundreds of people entering the supermarket.

First we need to consider the hardware. The diagram shown below shows the connection of a 7-segment display to Port B. Actually you would need a resistor in series with each LED segment to limit the current; a value of around 330Ω is typical. We will assume that the device is common-cathode, that is a 5V (logic 1) on a pin lights the segment and a 0V turns it off.

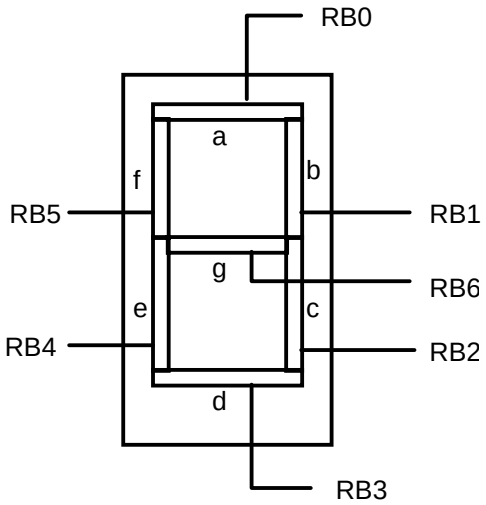


Figure 20: Adding a 7-segment display driven from Port B.

Looking at the software aspects, we need to take the number in the file called CENTURY (File 20h) and convert it to a code that will light the appropriate segments. For example, if the number in CENTURY is 02h (binary 00000010) then we need to light segments g, e, d, b and a above to give 2. The subroutine below simply adds on the number in the Working register to the low byte of the Program Counter, which is pointing to the first return instruction. Each `retlw` returns with the appropriate code in W. To illustrate the process, I have shown in red the situation if W contained the code 02h on entry. We have to assume that this number is never greater than nine otherwise the system will crash!

Self assessement Questions

1. How could you alter the subroutine so that a crash would never occur.
2. If the PC happened to be near a 256-page boundary, what would happen if the addition overflowed this boundary. For example, if PC = 1F6h. What could you do about it?

```
SVN_SEG   addwf   PCL          ; The key instruction adds n in W to PC
;                    gfedcba   ; The seven segments
          retlw   b'00111111' ; Code for zero
          retlw   b'00000110' ; Code for 1
          retlw   b'01011011' ; Code for 2
          retlw   b'01001111' ; Code for 3
          retlw   b'01100110' ; Code for 4
          retlw   b'01101101' ; Code for 5
          retlw   b'01111101' ; Code for 6
          retlw   b'00000111' ; Code for 7
          retlw   b'01111111' ; Code for 8
          retlw   b'01101111' ; Code for 9
```

Showing what happens when the contents of W are 02 on entry

Program 4: A 7-segment look-up table subroutine.

The first instruction adds the number in the Working register to that in the low byte of the Program Counter. Effectively this causes the Program Counter to skip forward n places at which it finds one of the ten return instructions. This causes execution to go back to the caller with the appropriate code in W.

Finally, we will repeat **Program 2** but now displaying the number of hundreds of people on a 7-segment display. The additional instructions are shown in red.

```
include   "p16f84.inc"
CENTURY   equ       20h    ; File 20h used to count hundreds of people

MAIN      bsf       STATUS,RP0 ; Move to Bank1 to access OPTION_REG
          movlw     b'00100100'; T0CS External, Prescaler, ratio 4
          movwf     OPTION_REG ; Send to Option register
          bcf       STATUS,RP0 ; Back to Bank 0

; Now set up Timer0 to -25 (actually to E7h) so that it will overflow
after 25 pulses (25 times 4 to give 100).  Also zero the batch count
          clrf      CENTURY     ; Start batch count at zero


LOOP      movf      CENTURY,w   ; Get the batch count
          call      SVN_SEG     ; Convert to 7-segment code
          movwf     PORTB       ; Send out to the display @ PORTB
          movlw     -d'25'      ; Put -25 in Timer0 (i.e. File 01)
          movwf     TMR0

OVERFLOW  btfss     INTCON,T0IF ; Check out the TMR0 overflow flag
          goto      OVERFLOW    ; Keep checking
          incf      CENTURY,f   ; Its set! So add one to the count
          bcf       INTCON,T0IF ; Clear this flag
          goto      LOOP        ; and start all over again
```

Seven-segment subroutine of **Program 4**
SVN_SEG

```
    end
```

Program 5: The complete people counter.

Self Assessment Questions

1. Our display can only indicate up to 900 people. Consider how you could extend this to 9900 people.

2. And to 99900?

Feeling tired? It's time for your last instruction. and it's called **sleep**. It sends the PIC MCU to sleep until something like the Watch Dog Timer, Reset or interrupt wakes it up. When a PIC MCU is asleep, it uses up less than a micro amp (compared to a few milli amps if the PIC MPU was just in a constant loop, for example), and all the outputs stay the same as they were before the chip fell asleep. The **Power Down (PD)** bit in the Status register is cleared on return from **sleep** and so the programmer can tell after an interrupt or Watchdog timeout that the return was after the execution of a **sleep** instruction; e.g. :

```
sleep           ; puts the PIC in a low current consuming mode
```

That's basically all that you will need to know to start programming, so ...

Remember...

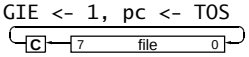
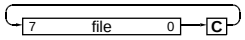
- Draw a flow chart or do a task list to clarify your thoughts!
- Be consistent.
- Keep a clear, easy to read, format.
- Explain all your instructions (without going over the top).
- Leave gaps in blocks of program.
- Take breaks.
- Your program must loop, it can't just end.
- Before you start the program, look at all the types of chips that are available and choose which one will be the best.
- Make a clear layout of what pins will go to which input and output devices.

Remember, anything's possible, just think it through carefully. One of the wonderful things about PIC Start is that there is no single way of achieving something (though the smaller programs are, the easier they are to understand and debug). You want a good program, your best effort and not necessarily the best solution.

Having Trouble Starting?

Just think out everything carefully. You should have a crystal clear idea of what you are going to do in order to achieve the circuit you wish to make. First, draw a flowchart of what your circuit would do. Then, make a pin layout, choose which pins you are going to connect your input and output devices to. Think of the methods or combinations of methods, that you will use to carry out certain parts of the circuit, because as much as possible should be pre-planned. As you write the program remember to take breaks and comment on all of your instructions, however trivial this may seem. Soon you will remember more and more instructions and pick up your own special techniques and style.

The PIC16F84 Instruction and Register set

14-bit op-code	16CXX Instruction	Mnemonic	Dest W	Dest F	CCR Z	CCR D	CCR C	Operation summary
11 1110 LLLL LLLL	ADD Literal to W	addlw LL	√		√	√	√	w <- w + #LL
00 0111 dfff ffff	ADD W and F	addwf f,d	√	√	√	√	√	d <- w + f
11 1110 LLLL LLLL	AND Literal to W	andlw LL	√		√	•	•	w <- w · #LL
00 0101 dfff ffff	AND W to F	andwf f,d	√	√	√	•	•	d <- w · f
01 00nn nfff ffff	Bit Clear File bit n	bcf f,n		√	•	•	•	f_n <- 0
01 01nn nfff ffff	Bit Set File bit n	bsf f,n		√	•	•	•	f_n <- 1
01 10nn nfff ffff	Bit Test File bit n & Skip if Clear	btfsc f,n		√	•	•	•	pc++ IF f_n == 0
01 11nn nfff ffff	Bit Test File bit n & Skip if Set	btfss f,n		√	•	•	•	pc++ IF f_n == 1
10 0aaa aaaa aaaa	CALL (jump to) subroutine	call aaa			•	•	•	TOS <- pc, pc <- aaa
00 0001 1fff ffff	CLeaR File	clrf f		√	√	•	•	f <- 00
00 0001 0000 0011	CLeaR Working register	clrw	√		√	•	•	d <- 00
00 0000 0000 0100	CLeaR Watch Dog Timer	clrwdt			•	•	•	wdt <- 00
00 1001 dfff ffff	COMplement File	comf f,d	√	√	√	•	•	d <- $\overline{f}$
00 0011 dfff ffff	DECrement File	decf f,d	√	√	√	•	•	d <- f--
00 1011 dfff ffff	DECrement File & Skip on Zero	decfsz f,d	√	√	•	•	•	d <- f--, pc++ IF f == 0
10 1aaa aaaa aaaa	GOTO (jump to) aaa	goto aaa			•	•	•	pc <- aaa
00 1010 dfff ffff	INCrement File	incf f,d	√	√	√	•	•	d <- f++
00 1111 dfff ffff	INCrement File & Skip on Zero	incfsz f,d	√	√	•	•	•	d <- f++, pc++ IF f == 0
11 1000 LLLL LLLL	Inclusive OR Literal to W	iorlw LL	√		√	•	•	w <- w + #LL
00 0100 dfff ffff	Inclusive OR W to F	iorwf f,d	√	√	√	•	•	d <- w + f
00 1000 dfff ffff	MOVe in File (load)	movf f,d	√	√	√	•	•	d <- f
11 0000 LLLL LLLL	MOVe Literal into W	movlw LL		√	•	•	•	w <- #LL
00 0000 1fff ffff	MOVe W out to File (store)	movwf f		√	•	•	•	f <- w
00 0000 0000 0000	No OPeration	nop			•	•	•	Do nothing
11 0100 LLLL LLLL	RETurn from subroutine with L in W	retlw	√		•	•	•	w <- #LL, pc <- TOS
00 0000 0000 1000	RETURN from subroutine	return			•	•	•	pc <- TOS
00 0000 0000 1001	RETurn From IntErrupt	retfie			•	•	•	GIE <- 1, pc <- TOS
00 1101 dfff ffff	Rotate Left File	rlf f,d	√	√	•	•	b_7	
00 1100 dfff ffff	Rotate Right File	rrf f,d	√	√	•	•	b_0	
00 0000 0110 0011	Sleep mode on	sleep			•	•	•	wdt <- 0, Clock off
11 1100 LLLL LLLL	SUB W from Literal	sublw LL	√		√	√	√	w <- #LL - w
00 0010 dfff ffff	SUBtract W from F	subwf f,d	√	√	√	√	√	d <- f - w
00 1110 dfff ffff	SWAP File nybbles	swapf f,d	√	√	•	•	•	d <- f[7:4] <--> f[3:0]
11 1010 LLLL LLLL	eXclusive OR Literal to W	xorlw LL	√		√	•	•	w <- w ⊕ #LL
00 0110 dfff ffff	eXclusive OR W to F	xorwf f,d	√	√	√	•	•	d <- w ⊕ f

√	: Flag operates in the normal manner	•	: Not affected	a...	: Address
d	: Destination; 0 = w, 1 = f	f...	: File register	f_n	: File bit n
L...	: Literal data	pc	: Program Counter	w	: Working register
wdt	: Watch Dog Timer/prescaler	TOS	: Top Of Stack	pc++	: Jump over next instruction
==	: Equivalent to	++	: Add one	--	: Subtract one
GIE	: Global Interrupt Enable mask	#	: Constant		

S.J. Katzen LaTeX 2_ε Version 2.1.0 November 13, 2001

pic_ins14.tex

PIC16F83/84 Special-Purpose Register file summary

File address	Name	7	6	5	4	3	2	1	0	Power-on Reset	All other Resets
Bank 0											
00*h*	INDF	Uses contents of this to address Data memory (not a physical register)									
01*h*	TMR0	8-bit real-time clock/counter								XXXX XXXX	UUUU UUUU
02*h*	PCL[1]	Lower-order 8 bits of the Program Counter								0000 00000	0000 0000
03*h*	STATUS[1]	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	**Z**	**DC**	**C**	0001 1XXX	000? ?UUU
04*h*	FSR	Indirect Data memory address pointer 0								XXXX XXXX	UUUU UUUU
05*h*	PORTA	—	—	—	RA4	RA3	RA2	RA1	RA0	—-X XXXX	—-U UUUU
06*h*	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0	XXXX XXXX	UUUU UUUU
08*h*	EEDATA	Data EEPROM Data register								XXXX XXXX	UUUU UUUU
09*h*	EEADR	Data EEPROM Address register								XXXX XXXX	UUUU UUUU
0A*h*	PCLATH	—	—	—	Write buffer for top 5 PC bits					——0 0000	——0 0000
0B*h*	INTCON	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000X	0000 000U
Bank 1											
80*h*	INDF	Uses contents of this to address Data memory (not a physical register)									
81*h*	OPTION	$\overline{RBPU}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	1111 1111
82*h*	PCL[1]	Lower-order 8 bits of the Program Counter								0000 00000	0000 0000
83*h*	STATUS[1]	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	**Z**	**DC**	**C**	0001 1XXX	000? ?UUU
84*h*	FSR	Indirect Data memory address pointer 0								XXXX XXXX	UUUU UUUU
85*h*	TRISA	—	—	—	Port A Direction Register					—-1 1111	—-1 1111
86*h*	TRISB	Port B Data Direction Register								1111 1111	1111 1111
88*h*	EECON1	Data EEPROM Data register								XXXX XXXX	UUUU UUUU
89*h*	EECON2	EEPROM Control register (not a physical register)									
8A*h*	PCLATH	—	—	—	Write buffer for top 5 PC bits					——0 0000	——0 0000
8B*h*	INTCON	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000X	0000 000U

X Not known

U Unchanged

? Value depends on whether a Watchdog Reset and if in Sleep mode before Reset.

— Unimplemented; reads as 0.

Note 1: Next instruction address if PIC in Sleep mode.

S.J. Katzen LaTeX 2ε Version 2.1.1 November 13, 2001

pic_ins14.tex