Sid Katzen

The Quintessential PIC Microcontroller

SPIN Springer's internal project number, if known

Engineering – Monograph (English)

November 8, 2000

Springer-Verlag
Berlin Heidelberg NewYork
London Paris Tokyo
Hong Kong Barcelona
Budapest

Contents

Appendices

List of Figures

List of Tables

List of Programs

PART I

The Fundamentals

This book is about microcontrollers (MCUs). These are digital engines modelled after the architecture of a stored-program computer and integrated on to a single very large-scale integrated circuit together with support circuitry, memories and peripheral interface devices. Although the MCU is often confused with its better known cousin the microprocessor in its role of the driving force of the ubiquitous personal computer, the vast majority of both microprocessors and microcontrollers are embedded into an assemblage of other digital components. The first microprocessors in the early 1970s were marketed as an alternative way of implementing digital circuitry. Here the task would be determined by a series of instructions encoded as binary code groups in read-only memory. This is more flexible than the alternative approach of wiring hardware integrated circuits in the appropriate manner. The microcontroller is simply the embodiment of this original role of the integrated computer.

We will look at embedded MCUs in a general digital processing context in Parts II and III. Here our objective is to lay the foundation for this material. We will be covering:

- *Digital code patterns.*
- *Binary arithmetic.*
- *Digital circuitry.*
- *Computer architecture and programming.*

This will by no means be a comprehensive review of the subject, but there are many other excellent texts in this area[1] which will launch you into greater depths.

[1] Such as S.J. Cahill's *Digital and Microprocessor Engineering*, 2nd edn., Prentice Hall, 1993.

CHAPTER 1

Digital Representation

To a computer or microprocessor, the world is seen in terms of patterns of digits. The **decimal** (or denary) system represents quantities in terms of the ten digits 0…9. Together with the judicious use of the symbols +, − and . any quantity in the range $\pm\infty$ can be depicted. Indeed non-numeric concepts can be encoded using numeric digits. For example the American Standard Code for Information Interchange (ASCII) defines the alphabetic (alpha) characters A as 65, B = 66…Z = 90 and a = 97, b = 98…z = 122 etc. Thus the string "Microprocessor" could be encoded as "77, 105, 99, 114, 111, 112, 114, 111, 99, 101, 115, 115, 111, 114". Provided you know the context, that is what is a pure quantity and what is text, then just about any symbol can be coded as numeric digits.[1]

Electronic circuits are not very good at storing and processing a multitude of different symbols. It is true that the first American digital computer, the ENIAC (Electronic Numerical Integrator And Calculator) in 1946 did its arithmetic in decimal[2] but all computers since handle data in **binary** (base 2) form. The decimal (base 10) system is really only convenient for humans, in that we have ten fingers.[3] Thus in this chapter we will look at the properties of binary digits, their groupings and processing. After reading it you will:

- Understand why a binary data representation is the preferred base for digital circuitry.
- Know how a quantity can be depicted in natural binary, hexadecimal and binary coded decimal.
- Be able to apply the rules of addition and subtraction for natural binary quantities.
- Know how to multiply by shifting left.
- Know how to divide by shifting right and propagating the sign bit.
- Understand the Boolean operations of NOT, AND, OR and XOR.

The information technology revolution is based on the manipulation, computation and transmission of digitized information. This informa-

[1] Of course there are lots of encoding standards, for example the 6-dot Braille code for the visually impaired.

[2] As did Babbage's mechanical computer of a century earlier.

[3] And ten toes, but base-20 systems are rare.

tion is virtually universally represented as aggregrates of *bi*nary digi*ts* (**bits**).[4] Most of this processing is effected using microprocessors, and it is sobering to reflect that there is more computing power in a singing birthday card than existed on the entire planet in 1950!

Binary is the universal choice for data representation, as an electronic switch is just about the easiest device that can be implemented using a transistor. Such 2-state switches are very small; they change state very quickly and consume little power. Furthermore, as there are only two states to distinguish between, a binary depiction is likely to be resistant to the effects of noise. The upshot of this is that both the packing density on a silicon chip and switching rate can be very high. Although a switch on its own does not represent much computing power; five million switches changing at 100 million times a second, manage to present at least a facade of intelligence!

The two states of a bit are conventionally designated **logic 0** and **logic 1** or just 0 & 1. A bit may be represented by two states of any number of physical quantities; for example electric current or voltage, light, pneumatic pressure. Most microprocessors use 0 V (or ground) for state 0 and 3 – 5 V for state 1, but this is not universal. For instance, the RS232 serial port on your computer uses nominally +12 V for state 0 and −12 V for state 1.

A single bit on its own can only represent two states. By dealing with groups of bits, rather more complex entities can be coded. For example the standard alphanumeric characters can be coded using 7-bit groups of digits. Thus the ASCII code for "Microprocessor" becomes:

1001101 1101001 1100011 1110010 1101111 1110000 1110010 1101111
1100011 1100100 1110011 1110011 1101111 1110010

Unicode is an extension of ASCII and with its 16-bit code groups is able represent characters from many languages and mathematical symbols.

The ASCII code is **unweighted**, as the individual bits do not signify a particular quantity; only the overall pattern has any significance. Other examples are the die code on gaming dice and 7-segment code of Fig. 6.6 on page 148. Here we will deal with **natural binary** weighted codes, where the position of a bit within the number field determines its value or weight. In an integer binary number the rightmost digit is worth $2^0 = 1$, the next left column $2^1 = 2$ and so on to the nth column which is worth 2^{n-1}. For example the decimal number one thousand nine hundred and ninety eight is represented as $1\times10^3 + 9\times10^2 + 9\times10^1 + 8\times10^0$ or 1998.

[4] The binary base is not a new fangled idea invented for digital computer; many cultures have used base 2 numeration in the past. The Harappān civilisation existed more than 4000 years ago in the Indus river basin. Found in the ruins of the Harappān city of Mohenjo-Daro, in the beadmakers' quarter, was a set of stone pebble weights. These were in ratios that doubled in the pattern, 1,1,2,4,8,16…, with the base weight of around 25g (≈ 1oz). Thus bead weights were expressed by digits which represented powers of 2; that is in binary.

Table 1.1: 7-bit ASCII characters.

MS nybble → / LS nybble	0*h* 000*b*	1*h* 001*b*	2*h* 010*b*	3*h* 011*b*	4*h* 100*b*	5*h* 101*b*	6*h* 110*b*	7*h* 111*b*
0*h* 0000*b*	NUL	DLE	SP	0	@	P	‘	p
1*h* 0001*b*	SOH	DC1	!	1	A	Q	a	q
2*h* 0010*b*	STX	DC2	"	2	B	R	b	r
3*h* 0011*b*	ETX	DC3	#	3	C	S	c	s
4*h* 0100*b*	EOT	DC4	$	4	D	T	d	t
5*h* 0101*b*	ENQ	NAK	%	5	E	U	e	u
6*h* 0110*b*	ACK	SYN	&	6	F	V	f	v
7*h* 0111*b*	BEL	ETB	’	7	G	W	g	w
8*h* 1000*b*	BS	CAN	(	8	H	X	h	x
9*h* 1001*b*	HT	EM	)	9	I	Y	i	y
A*h* 1010*b*	LF	SUB	*	:	J	Z	j	z
B*h* 1011*b*	VT	ESC	+	;	K	[	k	{
C*h* 1100*b*	FF	FS	,	<	L	\	l	|
D*h* 1101*b*	CR	GS	-	=	M	}	m	}
E*h* 1110*b*	SO	RS	.	>	N	^	n	~
F*h* 1111*b*	SI	US	/	?	O	_	o	DEL

In **natural binary** the same quantity is $1\times2^{10}+1\times2^9+1\times2^8+1\times2^7+1\times2^6+0\times2^5+0\times2^4+1\times2^3+1\times2^2+0\times2^1+1\times2^0$, or $11111001101b$. Fractional numbers may equally well be represented by columns to the right of the binary point using negative powers of 2. Thus $1101.11b$ is equivalent to 14.75. As can be seen from this example, binary numbers are rather longer than their decimal equivalent; on average a little over three times. Nevertheless, 2-way switches are considerably simpler than 10-way devices, so the binary representation is preferable.

An n-digit binary number can represent up to 2^n patterns. Most computers store and process groups of bits. For example the first micropro-

cessor, the Intel 4004, handled its data four bits (a **nybble**) at a time. Many current processors cope with blocks of 8 bits (a **byte**), 16 bits (a **word**), or 32 bits (a **long-word**). 64-bit (a **quad-word**) devices are on the horizon. These groupings are shown in Table 1.2. The names illustrated are somewhat de-facto, and variations are sometimes encountered.

As in the decimal number system, large binary numbers are often expressed using the prefixes k (kilo), M (mega) and G (giga). A binary kilo is $2^{10} = 1024$; for example 64 kbyte of memory. In an analogous way, a binary mega is $2^{20} = 1,048,576$; thus a 1.44 Mbyte floppy disk. Similarly a 2 Gbyte hard disk has a storage capacity of $2 \times 2^{30} = 2,147,483,648$ bytes. The former representation is certainly preferable.

Table 1.2: Some common bit groupings.

Bit (0–1)	(1 bit)	0–1
Nybble (0000–1111)	(4 bits)	0–15
Byte (0000 0000–11111 1111)	(8 bits)	0–255
Word (0000 0000 0000 0000–1111 1111 1111 1111)	(16 bits)	0–65,535
Long-word (0000 0000 0000 0000 0000 0000 0000 0000–1111 1111 1111 1111 1111 1111 1111 1111)	(32 bits)	0–4,294,967,295

Long binary numbers are not very human friendly. In Table 1.2, binary numbers were zoned into fields of four digits to improve readability. Thus the address of a data unit stored in memory might be 1000 1100 0001 0100 0000 1010*b*. If each group of four can be given its own symbol, 0…9 and A…F, as shown in Table 1.3, then the address becomes 8C140A*h*; a rather more manageable characterization. This code is called **hexadecimal**, as there are 16 symbols. Hexadecimal (base-16) numbers are a viable number base in their own right, rather than just being a convenient binary representation. Each column is worth $16^0, 16^1, 16^2 \ldots 16^n$ in the normal way.[5]

Binary Coded Decimal is a hybrid binary/decimal code extensively used at the input/output ports of a digital system (see Example 11.5 on page 298). Here each decimal digit is individually replaced by its 4-bit binary equivalent. Thus 1998 is coded as $(0001\ 1001\ 1001\ 1000)_{\mathrm{BCD}}$. This is very different from the equivalent natural binary code; even if it is represented by 0s and 1s. As might be expected, arithmetic in such

[5] Many scientific calculators, including that in the Accessories group under Windows 95, can do hexadecimal arithmetic.

Table 1.3: Different ways of representing the quantities decimal 0...20.

Decimal	Natural binary	Hexadecimal	Binary
00	00000	00	0000 0000
01	00001	01	0000 0001
02	00010	02	0000 0010
03	00011	03	0000 0011
04	00100	04	0000 0100
05	00101	05	0000 0101
06	00110	06	0000 0110
07	00111	07	0000 0111
08	01000	08	0000 1000
09	01001	09	0000 1001
10	01010	0A	0001 0000
11	01011	0B	0001 0001
12	01100	0C	0001 0010
13	01101	0D	0001 0011
14	01110	0E	0001 0100
15	01111	0F	0001 0101
16	10000	10	0001 0110
17	10001	11	0001 0111
18	10010	12	0001 1000
19	10011	13	0001 1001
20	10100	14	0010 0000

a hybrid system is difficult, and BCD is normally converted to natural binary at the system input and processing is done in natural binary before being converted back (see Program 5.9 on page 131).

The rules of arithmetic are the same in natural binary[6] as they are in the more familiar base 10 system, indeed any base-n radix scheme. The simplest of these is **addition**, which is a shorthand way of totalling quantities, as compared to the more primitive counting or incrementation process. Thus $2+4=6$ is rather more efficient than $2+1=3; 3+1=4; 4+1=5; 5+1=6$. However, it does involve memorizing the rules of addition.[7] In decimal this involves 45 rules, assuming that order is irrelevant; from $0+0=0$ to $9+9=18$. Binary addition is much simpler as it is covered by only three rules:

$$\begin{array}{ll} 0+0 & = 0 \\ \left.\begin{array}{l}0+1\\1+0\end{array}\right\} & = 1 \\ 1+1 & = 10 \quad (0 \text{ carry } 1) \end{array}$$

Based on these rules, the least significant bit (LSB) is totalized first, passing a **carry** if necessary to the next left column. The process ends with

[6] Sometimes called 8-4-2-1 code after the weightings of the first four lowest columns.

[7] Which you had to do way back in the mists of time in primary/elementary school!

the most significant bit (MSB) column, its carry being the new MSD of the sum. For example:

1			1		
0 1			2 6 3 1		
0 0 1			8 4 2 6 8 4 2 1		
96	Augend		1100000	Augend	
+ 37	Addend		+ 0100101	Addend	
1 1	Carries		1 1	Carries	
133	Sum		10000101	Sum	
(a) Decimal			*(b) Binary*		

Just as addition implements an up count, **subtraction** corresponds to a down count, where units are removed from the total. Thus $8 - 5 = 3$ is the equivalent of $8 - 1 = 7; 7 - 1 = 6; 6 - 1 = 5; 5 - 1 = 4; 4 - 1 = 3$.

The technique of decimal subtraction you are familiar with applies the subtraction rules commencing from LSB and working to the MSB. In any given column were a larger quantity is to be taken away from a smaller quantity, a unit digit is **borrowed** from the next higher column and given back after the subtraction is completed. Based on this borrow principle, the subtraction rules are given by:

$$
\begin{aligned}
0 - 0 &= 0 \\
{}^{1}0 - 1 &= 1 \quad \text{Borrowing 1 from the higher column} \\
1 - 0 &= 1 \\
1 - 1 &= 0
\end{aligned}
$$

For example:

1			6 3 1		
0 1			4 2 6 8 4 2 1		
96	Minuend		1100000	Minuend	
– 37	Subrahend		– 0100101	Subrahend	
1	Borrows		1 1 1 1 1 1	Borrows	
59	Difference		0111011	Difference	
(a) Decimal			*(b) Binary*		

Although this familiar method works well, there are several problems implementing it in digital circuitry.

- How can we deal with situations where the minuend is larger than the subtrahend?
- How can we distinguish between positive and negative quantities?
- Can a digital system's adder circuits be coerced into subtracting?

To illustrate these points, consider the following example:

37	Minuend	0100111	Minuend
– 96	Subtrahend	– 1100000	Subtrahend
41	Difference (–59)	1000111	Difference (–0111001)

(a) Decimal *(b) Binary*

Normally when we know that the when Minuend is greater than the Subtrahend, the two operands are interchanged and a minus sign is appended to the outcome; that is $-(\text{Subtrahend} - \text{Minuend})$. If we do not swap, as in (a) above, then the outcome appears to be incorrect. In fact 41 is correct, in that this is the difference between 59 (the correct outcome) and 100. 41 is described as the **10's complement** of 59. Furthermore, the fact that a borrow digit was generated from the MSD indicates that the difference is negative, and therefore appears in this 10's complement form. Converting from 10's complement decimal numbers to the 'normal' magnitude form is simply a matter of inverting each digit and then adding one to the outcome. A decimal digit is inverted by computing its difference from 9. Thus the 10's complement of 3941 is −6059:

$$\overline{3941} \Longrightarrow 6058; +1 = -6059$$

However, there is no reason why negative numbers should not remain in this complement form – just because we are not familiar with this type of notation.

The complement method of negative quantity representation of course applies to binary numbers. Here the ease of inversion ($0 \rightarrow 1; 1 \rightarrow 0$) makes this technique particularly attractive. Thus in our example above:

$$\overline{1000111} \Longrightarrow 0111000; +1 = -0111001$$

Again, negative numbers should remain in a **2's complement** form. This complement process is reversible. Thus:

$$\text{complement} \Longleftrightarrow \text{normal}$$

Signed decimal numeration has the luxury of using the symbols + and − to denote positive and negative quantities. A 2-state system is stuck with 1s and 0s. However, looking at the last example gives us a clue on how to proceed. A negative outcome gives a borrow back out to the highest column. Thus we can use this MSD as a **sign bit**, with 0 for + and 1 for −. This gives $1{,}1000111_b$ for −59 and $0{,}01110011_b$ for +59. Although for clarity the sign bit has been highlighted above using a comma delimiter, the advantage of this system is that it can be treated in all arithmetic processes in the same way as any other ordinary bit. Doing this, the outcome will give the correct sign:

```
 0,1100000  (+96)
 1,1011011  (−37)
 ———1———————
 0,0111011  (+59)
```

(a) Minuend less than subtrahend

```
 0,0100101  (+37)
 1,0100000  (−96)
 ———1———————
 1,1000101  (−59)
```

(b) Minuend greater than subtrahend

From this example we see that if negative numbers are in a signed 2's complement form, then we no longer have the requirement to implement hardware subtractors, as adding a negative number is equivalent to subtracting a positive number. Thus $\mathsf{A} - \mathsf{B} = \mathsf{A} + (-\mathsf{B})$. Furthermore, once numbers are in this form, the outcome of any subsequent processing will always remain 2's complement signed throughout.

There are two difficulties associated with signed 2's complement arithmetic. The first of these is **overflow**. It is possible that adding two positive or two negative numbers will cause overflow into the sign bit; for instance:

```
 0,1000  (+8)
 0,1011  (+11)
 ———1———————
 1,0011  (−13!!!)
```

(a) Sum of two +ve numbers gives −ve

```
 1,1000  (−8)
 1,0101  (−11)
1————————————
 0,1101  (+3!!!)
```

(b) Sum of two −ve numbers gives +ve

In (a) the outcome of $(+8) + (+11)$ is -13! The 2^4 numerical digit has overflowed into the sign position (actually, $10011b = 19$ is the correct outcome). Example (b) shows a similar problem for the addition of two signed negative numbers. Overflow can only happen if both operands have the *same* sign bits. Detection is then a matter of determining this situation with an outcome that differs. See Fig. 1.5 for a logic circuit to implement this overflow condition.

The final problem concerns arithmetic on signed operands with different sized fields. For instance:

```
 0,0011001  (+25)
      0,011  (+03)
 ———1———————
 ????
```

```
 0,0011001  (+25)
 0,0000011  (+03)
 ————————11—
 0,0011100  (+28)
```

(a) Extending a positive number

```
 0,0011001  (+25)
      1,101  (−03)
 ————————————
 ????
```

```
 0,0011001  (+25)
 1,1111101  (−03)
1—1—1111———1—
 0,0010110  (+22)
```

(b) Extending a negative number

Both the examples involve adding an 8-bit to a 16-bit operand. Where the former is positive, the data may be increased to 16 bits by padding with 0s. The situation is slightly less intuitive where negative data requires extension. Here the prescription is to extend the data by padding out with 1s. In the general case the rule is simply to pad out data by propagating the sign bit left. This technique is known as **sign extension**.

Multiplication by the nth power of two is simply implemented by shifting the data left n places. Thus 00101(5) << 01010(10) << 10100(20) multiplies 5 by 2^2, where the << operator is used to denote shifting left. The process works for signed numbers:

```
0,00000011  ( 3)      1,11111101  ( ¡3)        0,00000110  (3 x 2)
     <<                    <<                + 0,00011000  (3 x 8)
0,00000110  ( 6)      1,11111010  ( -6)       ------------
     <<                    <<                  0,00011110  (3 x 10 = 30)
0,00001100  (12)      1,11110100  (-12)
     <<                    <<
0,00011000  (24)      1,11101000  (-24)
```

(a) +3 x 8 = +24 *(b) −3 x 8 = −24* *(c) +3 x 10 = 30*

Should the sign bit change polarity, then a magnitude bit has overflowed. Some computers/microprocessors have a Arithmetic Shift Left process that signals this situation, as opposed to the standard Logic Shift Left used in unsigned number shifts.

Multiplication by non-powers of 2 can be implemented by a combination of shifting and adding. Thus as shown in (c) above, 3×10 is implemented as $(3 \times 8) + (3 \times 2) = (3 \times 10)$ or $(3 << 3) + (3 << 1)$.

In a similar fashion, division by powers of 2 is implemented by shifting right n places. Thus 1100(12) >> 0110(6) >> 0011(3) >> 0001.1(1.5). This process also works for signed numbers:

```
0,1111.000  (+15)      1,0001.000  (-15)          0001.1
     >>                    >>                1010 | 1111.0
0,0111.100  (+7.5)     1,1000.100  (-7.5)        -1010
     >>                    >>                   ------
0,0011.110  (+3.75)    1,1100.010  (-3.75)       0101
     >>                    >>                   -101.0
0,0001.111  (+1.875)   1,1110.001  (-1.875)     ------
                                                  000.0
```

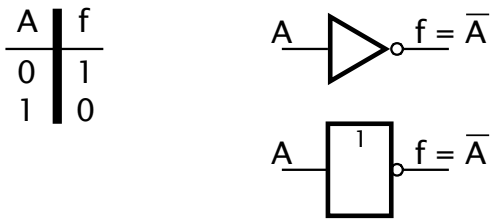
(a) +15/8 = 1.875 *(b) −15/8 = −1.875* *(c) 15/10 = 1.5*

Notice that rather than always shifting in 0s, the sign bit should be propagated in from the left. Thus positive numbers shift in 0s and negative numbers shift in 1s. This is known as **Arithmetic Shift Right** as opposed to **Logic Shift Right** which always shifts in 0s.

Division by non powers of 2 is illustrated in (c) above. This shows the familiar long division process used in decimal division. This is an

analagous process to the shift and add technique for multiplication, using a combination of shifting and subtracting.

Arithmetic is not the only way to manipulate binary patterns. George Boole[8] in the mid-19th century developed an algebra dealing with symbolic processing of logic propositions. This **Boolean algebra** deals with variables which can be true or false. In the 1930s it was realised that this mathematical system could equally well be used to analyze switching networks and thus binary logic systems. Here we will confine ourselves to looking at the fundamental logic operations of this switching algebra.



(a) Truth table *(b) Alternative logic symbols*

Fig. 1.1 The NOT operation.

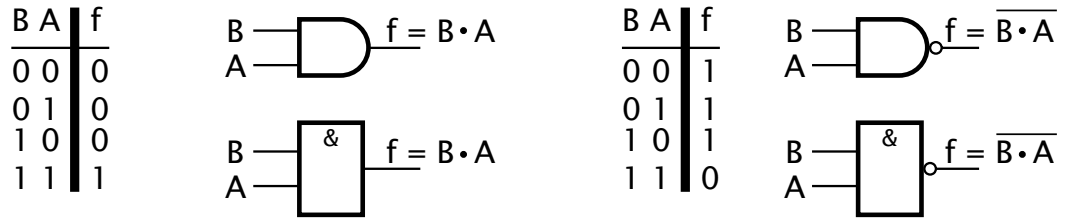
The inversion or **NOT** operation is represented by overscoring. Thus $f = \overline{A}$ states that the variable f is the inverse of A; that is if $A = 0$ then $f = 1$ and if $A = 1$ then $f = 0$. In Fig. 1.1(a) this transfer characteristic is presented in the form of a **truth table**. By definition, inverting twice returns a variable to its original state; thus $\overline{\overline{f}} = f$.[9]

Logic function implementations are normally represented in an abstract manner rather than as a detailed circuit diagram. The **NOT gate** is symbolized as shown in Fig. 1.1(b). The circle *always* represents inversion in a logic diagram, and is often used in conjunction with other logic elements, such as in Fig. 1.2(c).

The **AND operator** gives an *all or nothing* function. The outcome will only be true when *every* one of the *n* inputs are true. In Fig. 1.2 two input variables are shown, and the output is symbolized as $f = B \cdot A$, where $\cdot$ is the Boolean AND operator. The number of inputs is not limited to two, and in general $f = A(0) \cdot A(1) \cdot A(2) \cdots A(n)$. The AND operator is

[8] The first professor of mathematics at Queen's College, Cork.

[9] In days of yore when logic circuits were built out of discrete devices, such as diodes, resistors and transistors, problems due to sneak current paths were rife. In one such laboratory experiment the output lamp was rather dim, and the lecturer in charge suggested that two NOTs in series in a suspect line would not disturb the logic but would block off the unwanted current leak. On returning sometime later, the students complained that the remedy had had no effect. On investigation the lecturer discovered two knots in the offending wire – obviously not tied tightly enough!



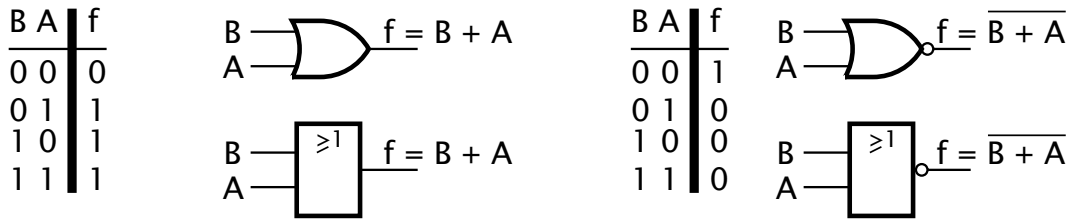
(a) Truth table *(b) Alternative logic symbols* *(c) NAND*

Fig. 1.2 The AND function.

sometimes called a logic product, as ANDing (cf. multiplying) any bit with logic 0 always yields a 0 output.

If we consider B as a control input and A as a stream of data, then consideration of the truth table shows that the output follows the data stream when B = 1 and is always 0 when B = 0. Thus the circuit can be considered to be acting as a valve, gating the data through on command. The term **gate** is generally applied to any logic circuit implementing a fundamental Boolean operator.

Most practical AND gate implementations have an inverting output. The logic of such implementations is NOT AND, or NAND for short, and is symbolized as shown in Fig. 1.2(c).



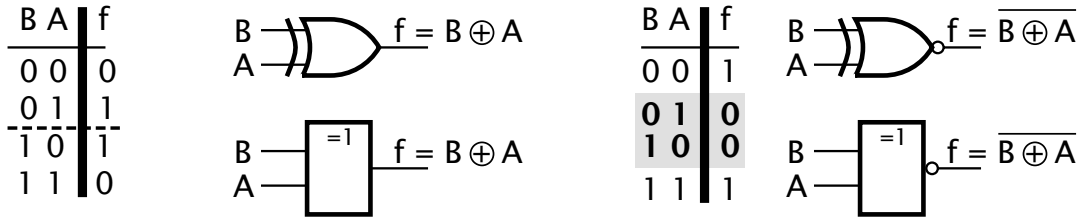
(a) Truth table *(b) Alternative logic symbols* *(c) NOR*

Fig. 1.3 The inclusive-OR operation.

The **inclusive-OR operator** gives an *anything* function. Here the outcome is true when *any* input or inputs are true (hence the ≥ 1 label in the logic symbol). In Fig. 1.3 two inputs are shown, but any number of variables may be ORed together. ORing is sometimes referred to as a logic sum, and the + used as the mathematical operator; thus f = B + A. In an analogous manner to the AND gate detecting all ones, the OR gate can be used to detect all zeroes. This is illustrated in Fig. 2.19 on page 35 where an 8-bit zero outcome brings the output of the NOR gate to 1.

Considering B as a control input and A as data (or vice versa), then from Fig. 1.3(a) we see that the data is gated through when B is 0 and inhibited (always 1) when B is 1. This is a little like the inverse of the AND function. In fact the OR function can be expressed in terms of AND using the duality relationship $\overline{A + B} = \overline{B} \cdot \overline{A}$. This states that the NOR function can be implemented by inverting all inputs into an AND gate.

AND, OR and NOT are the three fundamental Boolean operators. There is one more operation commonly available as an electronic gate; the **Exclusive-OR operator** (**XOR**). The XOR function is true if *only one* input is true (hence the =1 label in the logic symbol). Unlike the inclusive-OR, the situation where both inputs are true gives a false outcome.



(a) Truth table *(b) Alternative logic symbols* *(c) ENOR*

Fig. 1.4 The XOR operation.

If we consider B is a control input and A as data (they are fully interchangeable) then:

- When $B = 0$ then $f = A$; that is the output follows the data input.
- When $B = 1$ then $f = \overline{A}$; that is the output is the inverse of the data input.

Thus an XOR gate can be used as a programmable inverter.

Another useful property considers the XOR function as a logic differentiator. The XOR truth table shows that the gate gives a true output if the two inputs differ. Alternatively, the ENOR truth table of Fig. 1.4(c) shows a true output when the two inputs are the same. Thus an ENOR gate can be considered to be a 1-bit equality detector. The equality of two n-bit words can be tested by ANDing an array of ENOR gates (see Fig. 2.6 on page 23), each generating the function $\overline{B_k \oplus A_k}$; that is:

$$f_{B=A} = \sum_{k=0}^{n-1} \overline{B_k \oplus A_k}$$

As a simple example of the use of the XOR/XNOR gates, consider the problem of detecting sign overflow (see page 10). This occurs if both the sign bits of word B and word A are the same ($\overline{S_B \oplus S_A}$) AND the sign

bit of the outcome word C is not the same as either of these sign bits, say $S_B \oplus S_C$. The logic diagram for this detector is shown in Fig. 1.5 and implements the Boolean function:

$$(\overline{S_B \oplus S_A}) \cdot (S_B \oplus S_C)$$

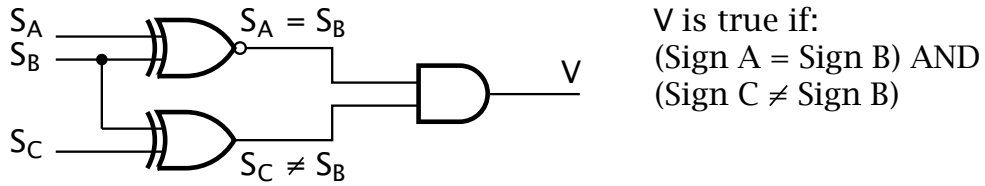


Fig. 1.5 Detecting sign overflow.

Finally, the XOR function can be considered as detecting when the number of true inputs are odd. By cascading $n + 1$ XOR gates, the overall parity function is true if the n-bit word has an odd number of ones. Some measure of error protection can be obtained by adding an additional bit to each word, so that overall the number of bits is odd. This oddness can be checked at the receiver and any deviation indicates corruption.

CHAPTER 2

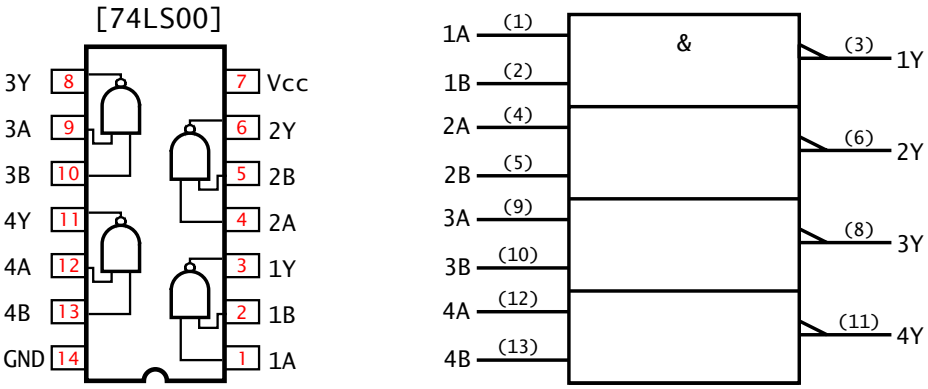
Logic Circuitry

We have noted that digital processing is all about transmission, manipulation and storage of binary word patterns. Here we will extend the concepts introduced in the last chapter as a lead into the architecture of the computer and microprocessor. We will look at some relevant logic functions, their commercial implementations and some practical considerations.

After reading this chapter you will:

- Understand the properties and use of active pull-up, open-collector and 3-state output structures.
- Appreciate the logic structure and function of the natural decoder.
- See how a MSI implementation of an array of ENOR gates can compare two words for equality.
- Understand how a 1-bit adder can be constructed from gates, and can be extended to deal with the addition of two n-bit words.
- Appreciate how the function of an ALU is so important to a programmable system.
- Be aware of the structure and utility of a read-only memory (ROM).
- Understand how two cross-coupled gates can implement a R S latch.
- Appreciate the difference between a D latch and D flip flop.
- Understand how an array of D flip flops or latches can implement a register.
- See how a serial connection of D flip flops can perform a shifting function.
- Understand how a D flip flop can act as a frequency divide by two, and how a cascade of these can implement a binary count.
- See how an ALU/PIPO register can implement an accumulator processor unit.
- Appreciate the function of a RAM.

The first integrated circuits, available at the end of the 1960s, were mainly NAND, NOR and NOT gates. The most popular family of logic functions was, and still is, the 74 series transistor transistor logic (TTL); introduced by Texas Instruments and soon copied by all the major major semiconductor manufacturers.



(a) DIL package *(b) ANSI/IEC logic symbol*

Fig. 2.1 The 74LS00 quad 2-I/P NAND package.

The 74LS00[1] comprises four 2-input NAND gates in a 14-pin package. The integrated circuit (IC) is powered with a 5 ± 0.25 V supply between V_{CC}[2] (usually about 5 V) and GND.The logic outputs are 2.4 – 5 V High and 0 – 0.4 V for Low. Most IC logic families require a 5 V supply, but 3 V versions are becoming available, and some CMOS implementations can operate with a range of supplies between 3 V and 15 V.

The 74LS00 IC is shown in Fig. 2.1(a) in its Dual In-Line (DIL) package. Strictly it should be described as a positive-logic quad 2-I/P NAND, as the electrical equivalent for the two logic levels 0 and 1 are Low (L is around ground potential) and High (H is around Vcc,[3] usually about 5 V). If the relationship 0 → H; 1 → L is used (negative logic) then the 74LS00 is actually a quad 2-I/P NOR gate. The ANSI/IEC[4] logic symbol of Fig. 2.1(b) denotes a Low electrical potential by using the polarity ◺ symbol. The ANSI/IEC NAND symbol shown is thus based on the *real* electrical operation of the circuit. In this case the logic coincides with a

[1] The LS stands for "Low-power Schottky transistor". There are very many other versions, such as ALS (Advanced LS), AS (Advanced Schottky) and HC (High-speed Complementary metal-oxide transistor – CMOS). These family variants differ in speed and power consumption, but for a given number designation have the same logic function and pinout.

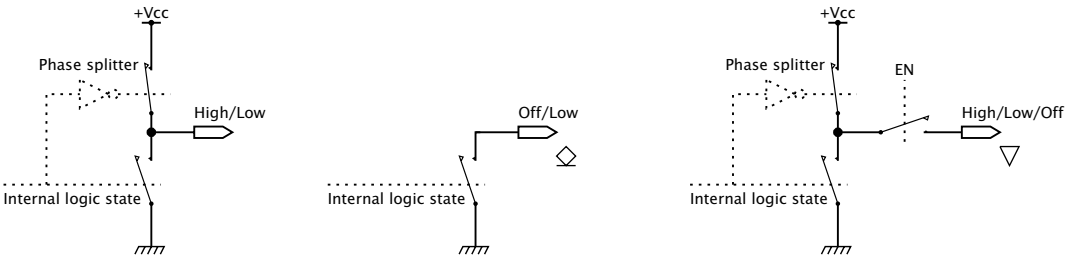
[2] For historical reasons the positive supply on logic ICs are usually designated as V_{CC}; the C referring to a bipolar's transistor Collector supply. Similarily field-effect circuitry sometimes use the designation V_{DD} for Drain voltage. The zero reference pin is normally designated as the ground point (GND), but sometimes the V_{EE} (for emitter) or V_{SS} (for Drain) label is employed.

[3] For historical reasons the positive supply on logic ICs are usually designated as V_{CC}; the C referring to a bipolar's transistor Collector supply. Similarily field-effect circuitry sometimes use the designation V_{DD} for Drain voltage. The zero reference pin is normally designated as the ground point (GND), but sometimes the V_{EE} (for emitter) or V_{SS} (for Drain) label is employed.

[4] The American National Standards Institution/International Electrotechnical Commission.

positive-logic NAND function. The & operator shown in the top block is assumed applicable to the three lower gates.

The output structure of a 74LS00 NAND gate is **active pull-up**. Here both the High and Low states are generated by connection via a low-resistance switch to Vcc or GND respectively. In Fig. 2.2(a) these switches are shown for simplicity as metallic contacts, but they are of course transistor derived.



(a) Push/pull (Totem-pole) *(b) Open-collector (open-drain)* *(c) Three-state*

Fig. 2.2 Output structures.

Logic circuits, such as the 74LS00, change output state in around 10 nanoseconds.[5] To be able to do this, the capacitance of any interconnecting conductors and other logic circuits' inputs must be rapidly discharged. Mainly for this reason, active pull-up (sometimes called totem-pole) outputs are used by most logic circuits. There are certain circumstances where alternative output structures have some advantages. The **open-collector** (or open-drain) configuration of Fig. 2.2(b) provides a 'hard' Low state, but the High state is in fact an open-circuit. The High-state voltage can be generated by connecting an external resistor to either Vcc or indeed to a different power rail. Non-orthodox devices, such as relays, lamps or light-emitting diodes, can replace this pull-up resistor. The output transistor is often rated with a higher than usual current and/or voltage rating for such purposes.

The application of most interest to us here is illustrated in Fig. 2.3. Here four open-collector gates share a *single* pull-up resistor. Note the use of the ◇ symbol to denote an open-collector output. Assume that there are four peripheral devices, any of which may wish to attract the attention of the processor (eg. computer or microprocessor). If this processor has only one Attention pin, then the four Signal lines must be **wire-ORed** together as shown. With all Signals inactive (logic 0) the outputs of all buffer NOT gates are off (state H), and the party line is pulled up to +V by RL. If *any* Signal line is activated (logic 1), as in Sig_1, then the output of the corresponding buffer gate goes hard Low. This pulls

[5] A nanosecond is 10^{-9} s, so 100,000,000 transitions each second is possible.

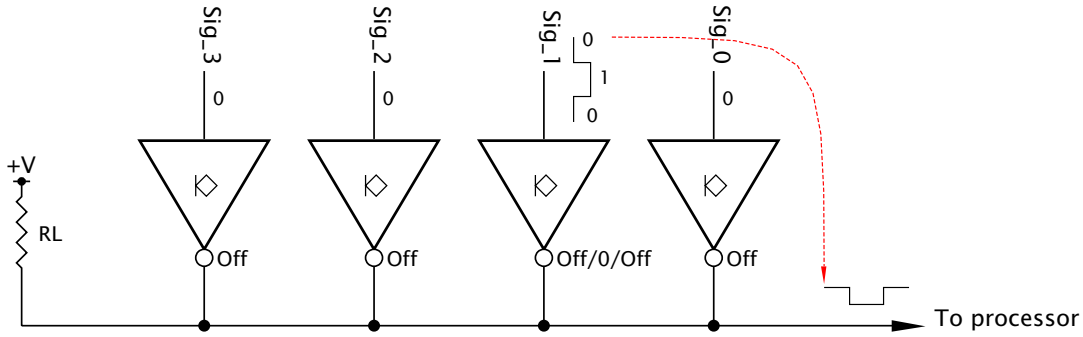


Fig. 2.3 Open-collector buffers driving a party line.

the party line Low, irrespective of the state of the other signal lines, and thus interrupts the processor.

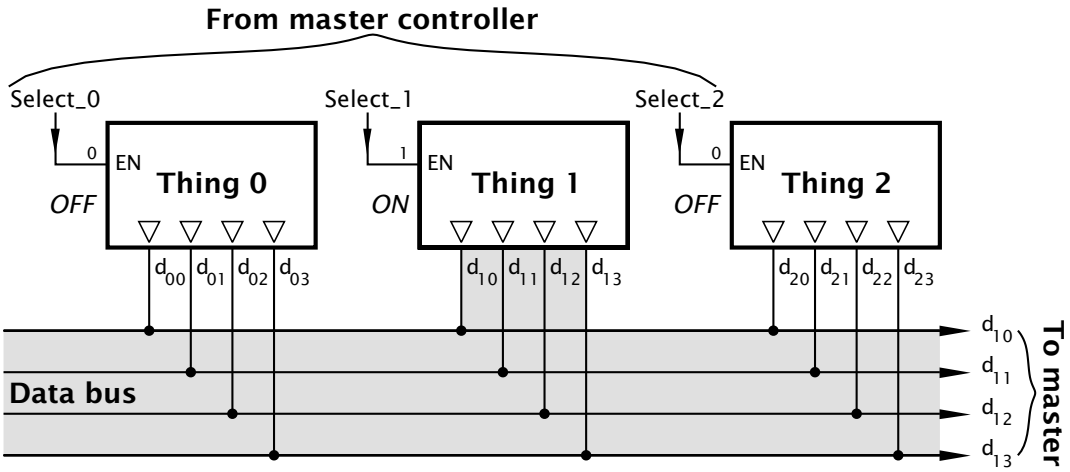


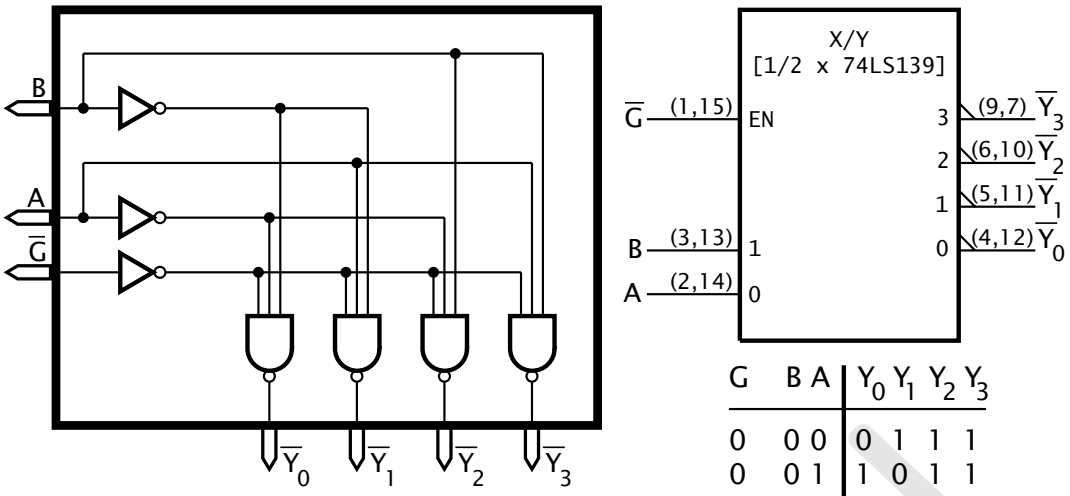
Fig. 2.4 Sharing a bus.

The **three-state** structure of Fig. 2.2(c) has the properties of both the preceeding output structures. When enabled, the two logic states are represented in the usual way by high and low voltages. When disabled, the output is open circuit irrespective of the activities of the internal logic circuitry and any change in input state. A logic output with this three-state is indicated by the ▽ symbol.

As an example of the use of this structure, consider the situation depicted in Fig. 2.4. Here a master controller wishes to read one of several devices, all connected to this master over a set of party lines. As this data highway or **Data bus** is a common resource, so only the selected device can be allowed access to the bus at any one time. The access has to be withdrawn immediately the data has been read, so that another device

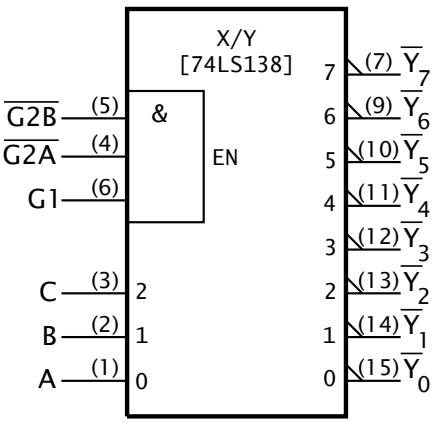
can use the resource. As shown in the diagram, each Thing connected to the bus outputs, designated by the $\bigtriangledown$ symbol. When selected, *only* the active logic levels will drive the bus lines. The 74LS244 octal (×8) 3-state (sometimes called tri-state or TRIS) buffer has high-current outputs (designated by the $\triangleright$ symbol) specifically designed to charge/discharge the capacitance associated with long bus lines.

Integrated circuits with a complexity of up to 12 gates are categorised as Small-Scale Integration (SSI). Gate counts upwards to 100 on a single IC are Medium-Scale Integration (MSI), up to 1000 are known as Large-Scale Integration (LSI) and over this, Very Large-Scale Integration (VLSI). Memory chips and microprocessors are examples of this latter category.



G	B	A	Y_0	Y_1	Y_2	Y_3
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0
1	X	X	1	1	1	1

(a) The 74LS139 dual 2- to 4-line decoder.



EN	C	B	A	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
0	0	0	0	0	1	1	1	1	1	1	1
0	0	0	1	1	0	1	1	1	1	1	1
0	0	1	0	1	1	0	1	1	1	1	1
0	0	1	1	1	1	1	0	1	1	1	1
0	1	0	0	1	1	1	1	0	1	1	1
0	1	0	1	1	1	1	1	1	0	1	1
0	1	1	0	1	1	1	1	1	1	0	1
0	1	1	1	1	1	1	1	1	1	1	0
1	X	X	X	1	1	1	1	1	1	1	1

(b) The 74LS138 3- to 8-line decoder

Fig. 2.5 The 74LS138 and ’139 MSI natural decoders.

The NAND gate networks shown in Fig. 2.5 are typical MSI-complexity ICs. Remembering that the output of a NAND gate is logic 0 only when *all* its inputs are logic 1 (see Fig. 1.2(c) on page 13) then we see that for any combination of the *Select* inputs B A (2^1 2^0) in Fig. 2.5(a) only *one* gate will go to logic 0. Thus output $\overline{Y_2}$ will be activated when B A = 10. The associated truth table shows the circuit *decodes* the binary address B A so that address n selects output $\overline{Y_n}$. The 74LS139 is described as a dual 2 to 4-line **natural decoder**. Dual because there are two such circuits in the one chip. The symbol X/Y denotes converting code X (natural binary) to code Y (unary – one of n). The Enable input $\overline{G}$ is connected to all gates in parallel. Thus the decoder function only operates if $\overline{G}$ is Low (logic 0). If $\overline{G}$ is High, then irrespective of the state of B A (the X entries in the truth table denote a 'don't care' situation) all outputs remain deselected – logic 1. An example of the use of the 74LS139 is given in Fig. 2.23.

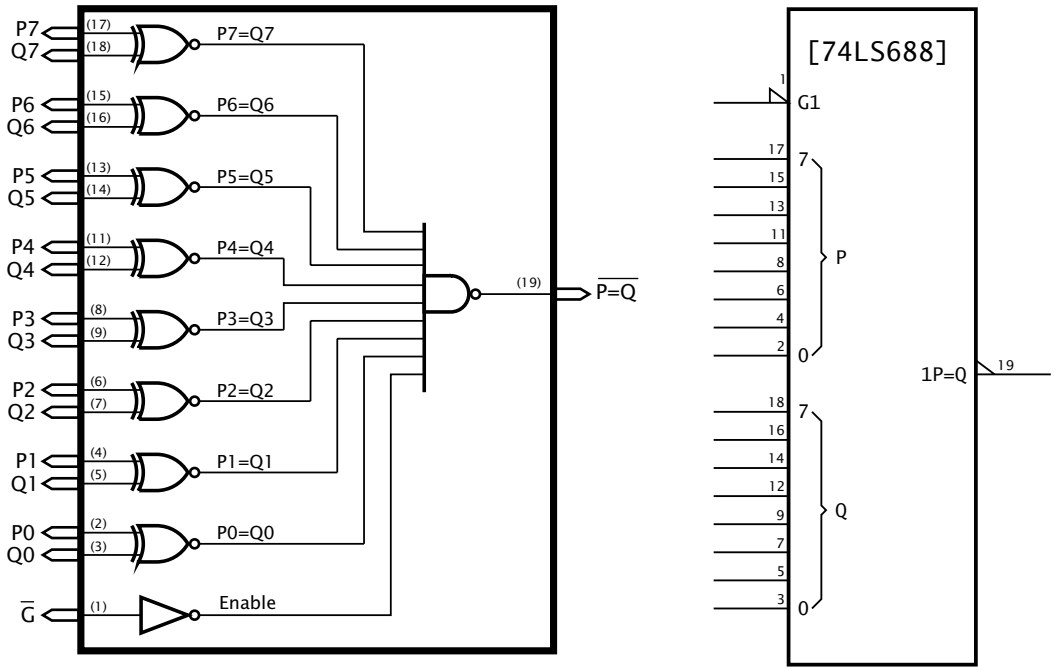
The 74LS138 of Fig. 2.5(b) is similar, but implements a 3 to 8-line decoder function. The state of the three address lines C B A (2^2 2^1 2^0) n selects one only of the eight outputs $\overline{Y_n}$. The 74LS138 has three Gate inputs which generate an internal Enable signal $\overline{G2B} \cdot \overline{G2A} \cdot G1$. Only if both $\overline{G2A}$ and $\overline{G2B}$ are Low and G1 is High will the device be enabled. The 74LS138 is used in Fig 11.10 on page 287 to decode microcontroller port lines to enable several devices to communicate to the one port.

A large class of ICs implement arithmetic operations. The gate array illustrated in Fig. 2.6 detects when the 8-bit byte P7...P0 is identical to the byte Q7...Q0. Eight ENOR gates each give a logic 1 when its two input bits Pn, Qn are identical, as described on page 14. Only if *all* eight bit pairs are the same, will the output NAND gate go Low. The 74LS688 **Equality comparator** also has a direct input $\overline{G}$ into this NAND gate, acting as an overall Enable signal.

The ANSI/IEC logic symbol, shown in Fig. 2.6(b) uses the COMP label to denote the arithmetic comparator function. The output is prefixed with the numeral 1, indicating that its operation P=Q is dependent on any input qualifying the same numeral; that is G1. Thus the active-Low Enable input G1 gates the active-Low output, 1P=Q.

One of the first functions beyond simple gates to be integrated into a single IC was that of addition. The truth table of Fig. 2.7(a) shows the Sum (S) and Carry-Out (C_1) resulting from the addition of the two bits A and B and any Carry-In (C_0). For instance row 6 states that adding two 1s with a Carry-In of 0 gives a Sum of 0 and a Carry-Out of 1 ($1 + 1 + 0 = {}^1 0$). To implement this row we require to detect the pattern 1 1 0; that is $A \cdot B \cdot \overline{C_0}$; which is gate 6 in the logic diagram. Thus we have by ORing all applicable patterns together for each output:

$$\begin{aligned} S &= (\overline{A} \cdot \overline{B} \cdot C_0) + (\overline{A} \cdot B \cdot \overline{C_0}) + (A \cdot \overline{B} \cdot \overline{C_0}) + (A \cdot B \cdot C_0) \\ C_1 &= (\overline{A} \cdot B \cdot C_0) + (A \cdot \overline{B} \cdot C_0) + (A \cdot B \cdot \overline{C_0}) + (A \cdot B \cdot C_0) \end{aligned}$$



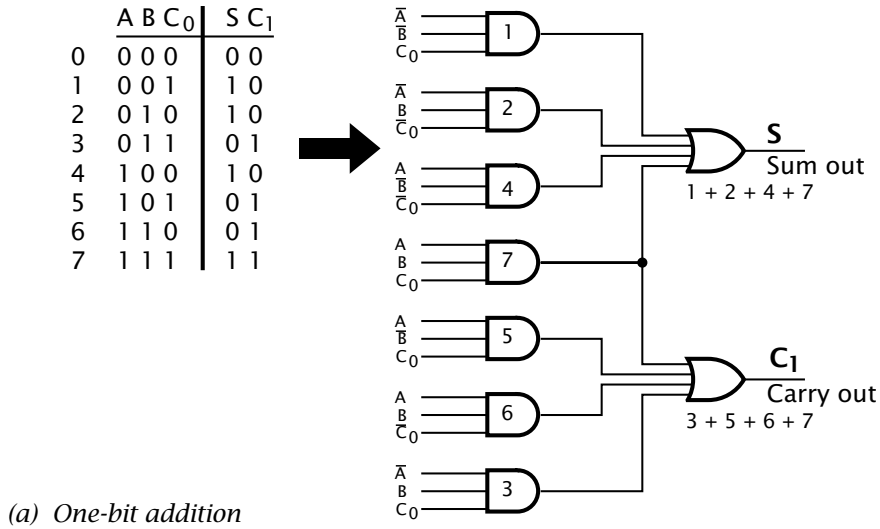
(a) Logic function *(b) ANSI/IEC logic symbol*

Fig. 2.6 The 74LS688 octal equality detector.

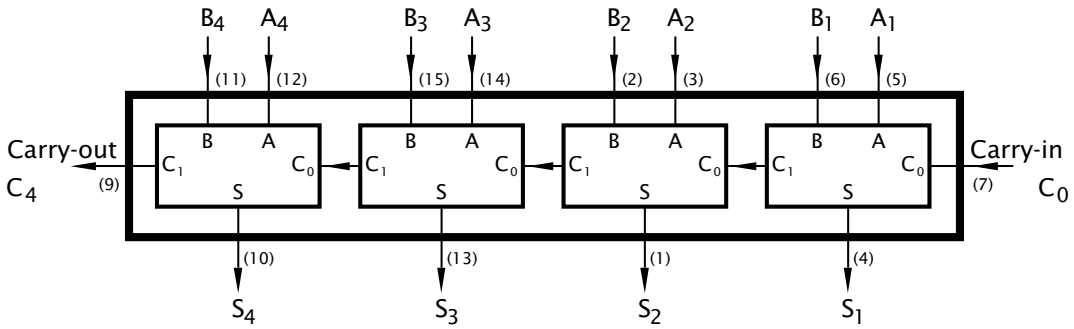
Using such a circuit for *each* column of a binary addition, with the Carry-Out from column $k-1$ feeding the Carry-In of column k means that the addition of any two n-bit words can be simultaneously implemented. As shown in Fig. 2.7(b), the 74LS283 adds two 4-bit nybbles in 25 ns. In practice the final Carry-Out C_4 is generated using additional circuitry to avoid the delays inherent on the carries rippling though each stage from the least to the most significant digit. n 74LS283s can be cascaded to implement addition for words of $4 \times n$ width. Thus two 74LS283s perform a 16-bit addition in 45 ns; the extra time being accounted for by the carry propagation between the two units.

Adders can of course be coaxed into subtraction by inverting the minuend and adding one, that is 2's complementation. An Adder/Subtractor circuit could be constructed by feeding the minuend word through an array of XOR gates acting as programmable inverters (see page 14). The Mode line $\overline{\text{Add/Sub}}$ in Fig. 2.8 that controls these inverters also feeds the Carry-In, effectively adding one when in the Subtract mode.

Extending this line of argument leads to the **Arithmetic Logic Unit** (**ALU**). An ALU is a circuit which can undertake a selection of arithmetic and logic processes on input data as controlled by Mode inputs. The 74LS382 in Fig. 2.9 processes two 4-bit operands in eight ways, as controlled by the three Select bits $S_2\,S_1\,S_0$ and tabulated in Fig. 2.9(a). Besides



(a) One-bit addition



(b) The 74LS283 four-bit adder

Fig. 2.7 Addition.

addition and subtraction, the logic operations of AND, OR and XOR are supported. The 74LS382 even generates the 2's complement overflow function (see page 10).

As we shall see, the ALU is the heart of the computer and microprocessor architectures. By feeding the Select inputs with a series of mode words, a program of operations can be performed by the ALU. Such **operation codes** are stored in an external memory, and are accessed sequentially by the computer's control circuits.

Sequences of program operation codes are normally stored in an LSI Read-Only Memory (ROM). Consider the architecture illustrated in Fig. 2.10. This is essentially a 3 to 8-line decoder driving an 8×2 array of diodes. The 3-bit address selects only row n for each input combination n. If a diode is connected to this row, then it conducts and brings the appropriate column Low. The inverting 3-state output buffer consequently gives a High for each connected diode and Low where the link is broken. The pattern of diode links then defines the output code for each input. For illustrative purposes, the structure has been programmed to

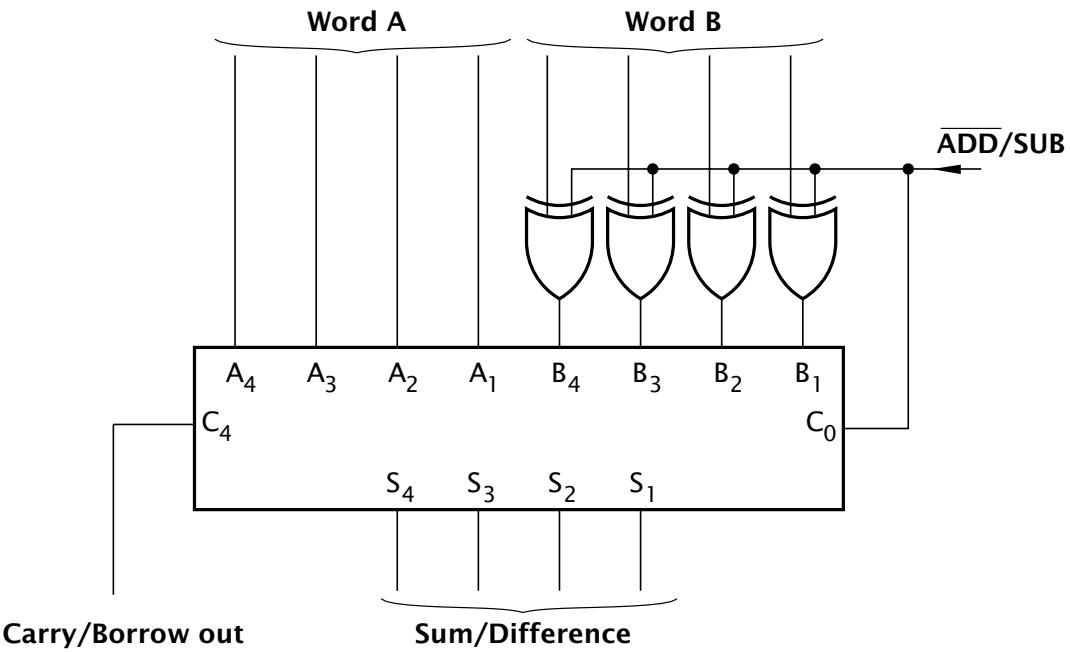
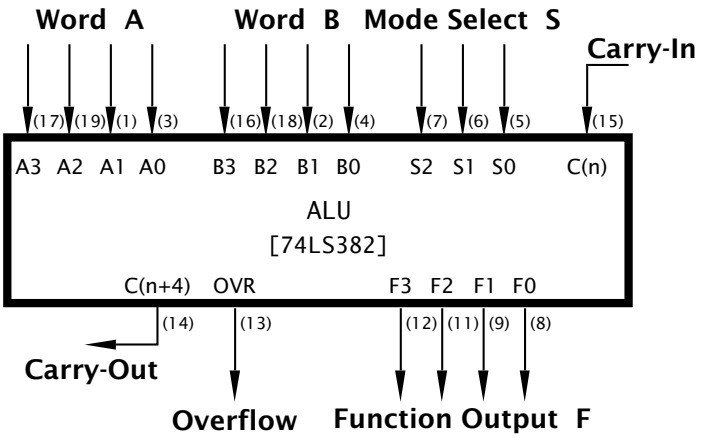


Fig. 2.8 Implementing a programmable adder/subtractor.

$S_2S_1S_0$	Operation	
0 0 0	Clear	(F=0000)
0 0 1	Subtract	(B-A)
0 1 0	Subtract	(A-B)
0 1 1	Add	(A+B)
1 0 0	EOR	(A ⊕ B)
1 0 1	OR	(A + B)
1 1 0	AND	(A • B)
1 1 1	Preset	(F=1111)

(a) Function table



(b) Logic diagram/pinning

Fig. 2.9 The 74LS382 ALU.

implement the 1-bit full adder of Fig. 2.7(a), but *any* two functions of three variables can be generated.

The diode matrix look-up table shown here is known as a **Read-Only Memory** (**ROM**), as its 'memory' is in the diode pattern, which is programmed in when the device is manufactured. Early devices, which were typically decoder/32 × 8 matrices, usually came in user-programmable versions in which the links were implemented with fusible links. By using

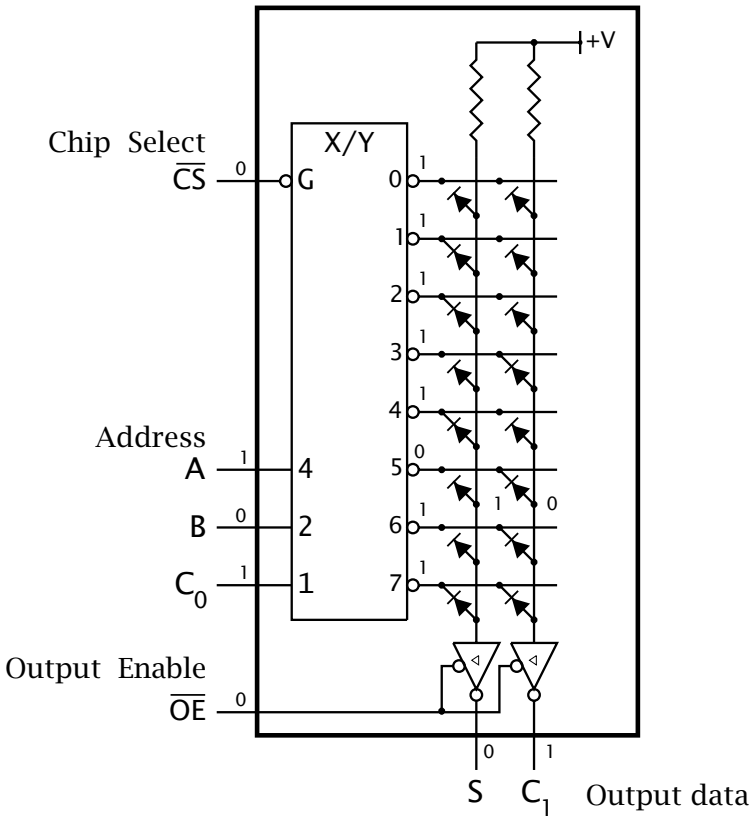
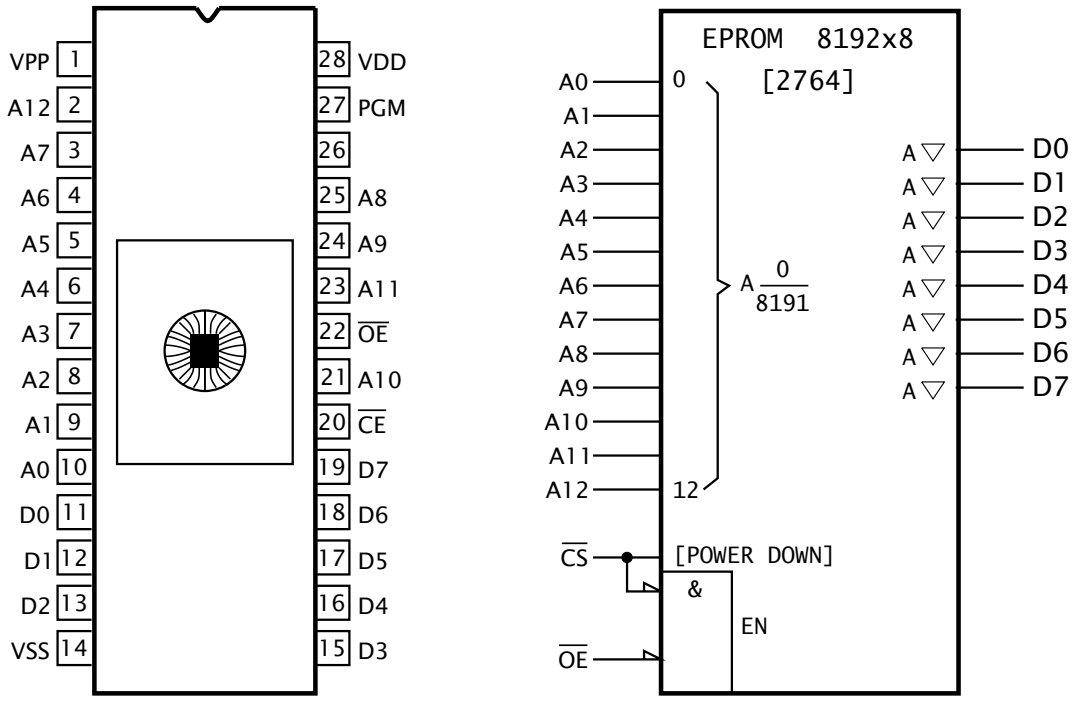


Fig. 2.10 A ROM-implemented 1-bit adder.

a high voltage, a selection of diodes could be taken out of contact. Such devices are called **Programmable ROMs** (**PROMs**).

Fuses are messy when implementing the larger sizes of VLSI PROM necessary to store computer programs. For example, the 27C64 PROM shown in Fig. 2.11 has the equivalent of 65,536 fuse/diode pairs, and this is a relatively small device capable of storing 8192 bytes of memory. The 27C64 uses electrical charge on the floating gate of a metal-oxide field-effect transistor (MOSFET) as the programmable link, with another MOSFET to replace the diode. Charge can be tunnelled onto this isolated gate by, again, using a high voltage. Once on the gate, the electric field keeps the link MOSFET conducting. This charge takes many decades to leak away, but this can be dramatically reduced to about 30 minutes by exposure to intensive ultra-violet radiation. For this reason the 27C64 is known as an **Erasable PROM** (**EPROM**). When an EPROM is designed for reusability, a quartz window is integrated into the package, as shown in Fig. 2.11. Programming is normally done externally with special equipment, known as PROM programmers, or colloquially as PROM blasters. Versions without windows are referred to as One-Time Programmable (OTP) ROMs, as they cannot easily be erased once programmed. They



(a) Dual in-line package *(b) Logic symbol*

Fig. 2.11 The 2764 Erasable PROM.

are however, much cheaper to produce and are thus suitable for small to medium-scale production runs.

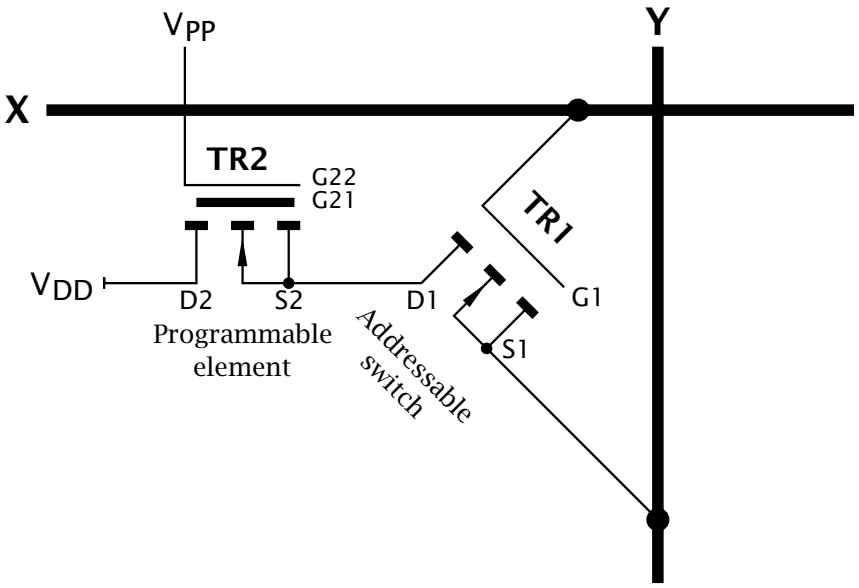


Fig. 2.12 Floating-gate MOSFET link

Figure 2.12 shows a simplified representation of such a floating-gate MOSFET link. The cross-point device is a metal-oxide enhancement n-channel field-effect transistor TR1, rather than a diode. This MOSFET has its gate G1 connected to the X line and its source S1 to the Y line. If its drain D1 were connected to the positive supply and the X line is selected (positive), then the Y line too becomes positive (positive-logic 1) as TR1 is conducting (switch is on). However, if TR1 is disconnected from V_{DD} then it does not conduct and the output on the Y line is logic 0. Transistor TR2 is in series with V_{DD} and thus acts as the programmable element. Transistor TR2 has an extra unconnected gate buried in the silicon dioxide insulation layer. Normally there is no charge on this gate and TR2 is off. If the programming voltage V_{PP} is pulsed high to typically 20 – 25 V, negative charges tunnel across the extemely thin insulation surrounding the buried gate. This permanently turns TR2 on and thus connects TR1 to its supply. This shows up as a logic 1 on the Y line when selected by the internal memory decoder.

This charge remains more or less permanently on the buried gate until it is exposed to ultra-violet light. The high-energy light photons knock electrons (negative charges) out of the buried (floating) gate[6] effectively discharging in around 20 minutes and wiping out all stored information. There are PROM structures which can be erased electrically, often in situe in the circuit. These are known variously as Electrically-Erasable PROMs (EEPROMs) or flash memories. In the former case a large negative pulse at V_{PP} causes the captured electrons on the buried gate to tunnel back out. Generally the negative voltage is generated on the chip, which saves having to provide an additional external supply. The **flash** variant of EEPROM relies on hot electron injection rather than tunneling to charge the floating gate. The geometry of the cell is approximately half the size of a conventional EEPROM cell which increases the memory density. Programming voltages are also somewhat lower.

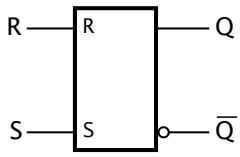
Most modern EPROM/EEPROMs are fairly fast, taking around 150 ns to access and read. Programming is slow, at perhaps 10 ms per word, but this is an infrequent activity. Flash EEPROM programs around 1000 times faster, in around 10 μs per cell.

All the circuits shown thus far are categorised as **combinational logic**. They have no memory in the sense that the output simply depends only on the present input, and not the sequence of events leading up to that input. Logic circuits, such as latches, counters, registers and read/write memories are described as **sequential logic**. Their output not only depends on the current input, but the sequence of prior inputs.

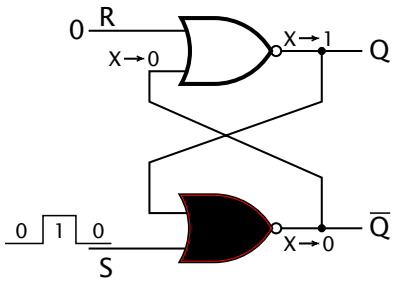
[6] This is called the Einstein effect. Einstein was awarded his Nobel prize for this discovery and not for his theories of relativity, as these were considered too revolutionary!

R S	Q	
0 0	Q	(no change)
0 1	1	(set)
1 0	0	(reset)

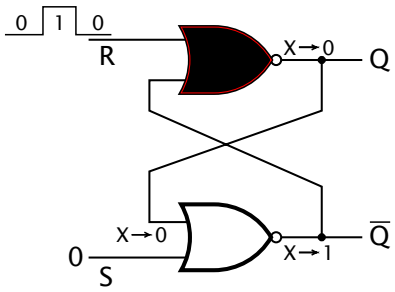
(a) Defining RS latch truth table



(b) Logic symbol with true/complement outputs



(c) Setting the latch



(d) Resetting the latch

Fig. 2.13 The R S latch.

Consider a typical door bell push-switch. When you press such a switch the bell rings, and it stops as soon as you release it. This switch has no memory.

Compare this with a standard light switch. Set the switch and the light comes on. Moreover it remains on when you remove the stimulus (usually your finger!). To turn the light off you must reset the switch. Again it remains off when the input is taken away. This type of switch is known as a **bistable**, as it has two stable states. Effectively it is a 1-bit memory cell, that can store either an on or off state indefinitely.

A read-write memory, such as the 6264 device of Fig. 2.24, implements each bistable cell using two cross-coupled transistors. Here we are not concerned with this microscopic view. Instead, consider the two cross-coupled NOR gates of Fig. 2.13. Remembering from Fig. 1.3(c) on page 13 that any logic 1 into a NOR gate will always give a logic 0 output irrespective of the state of the other inputs, allows us to analyse the circuit:

- If the S input goes to 1, then output $\overline{Q}$ goes to 0. Both inputs to the top gate are now 0 and thus output Q goes to 1. If the S input now goes back to 0, then the lower gate remains 0 (as the Q feedback is 1) and the top gate output also remains unaltered. Thus the latch is *set* by pulsing the S input.
- If the R input goes to 1, then output Q goes to 0. Both inputs to the bottom gate are now 0 and thus output $\overline{Q}$ goes to 1. If the R input now goes back to 0, then the upper gate remains 0 (as the $\overline{Q}$ feedback is 1) and the bottom gate output also remains unaltered. Thus the latch is *reset* by pulsing the R input.

In the normal course of events – that is assuming that the R and S inputs are not both active at the same time[7] then the two outputs are always complements of each other, as indicated by the logic symbol of Fig. 2.13(b).

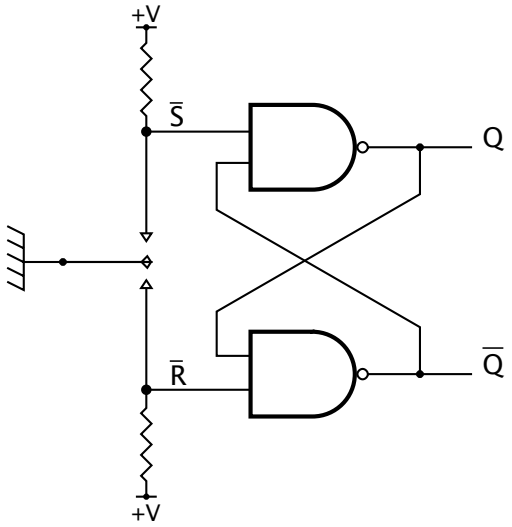


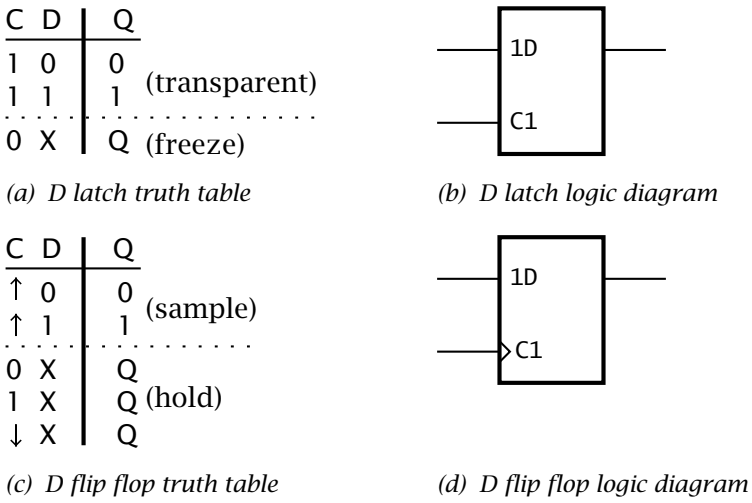
Fig. 2.14 Using a $\overline{R}\,\overline{S}$ latch to debounce a switch.

There are many bistable implementations. For example, replacing the NOR gates by NAND gives a $\overline{R}\,\overline{S}$ latch, where the inputs are active on a logic 0. The circuit illustrated in Fig. 2.14 shows such a latch used to debounce a mechanical switch. Manual switches are frequently used as inputs to logic circuits. However, most metallic contacts will bounce off the destination contact many times over a period of several tens of milliseconds before settling. For instance, using a mechanical switch to interrupt a computer/microprocessor will give entirely unpredictable results.

In Fig. 2.14, when the switch is moved up and hits the contact the latch is set. When the contact is broken, the latch remains unchanged, provided that the switch does not bounce all the way back to the lower contact. The state will remain Set no matter how many bounces occur. By symmetry, the latch will reset when the switch is moved to the bottom contact, and remain in this Reset state on subsequent bounces.

The **D latch** is an extension to the R S latch, where the output follows the D (Data) input when the C (Control) input is active (logic 1 in our example) and freezes when C is inactive. The D latch can be considered

[7] If they where, then both Q and $\overline{Q}$ go to 0. On relaxing the inputs, the latch will end up in one of its stable states, depending on the relaxation sequence. The response of a latch to a simultaneous Set and Reset is not part of the latch definition, shown in Fig. 2.13(a), but depends on its implementation. For example, trying to turn a light switch on and off together could end in splitting it in two!



(a) D latch truth table *(b) D latch logic diagram*

(c) D flip flop truth table *(d) D flip flop logic diagram*

Fig. 2.15 The D latch and flip flop.

to be a 1-bit memory cell where the datum is retained at its value at the end of the sample pulse.

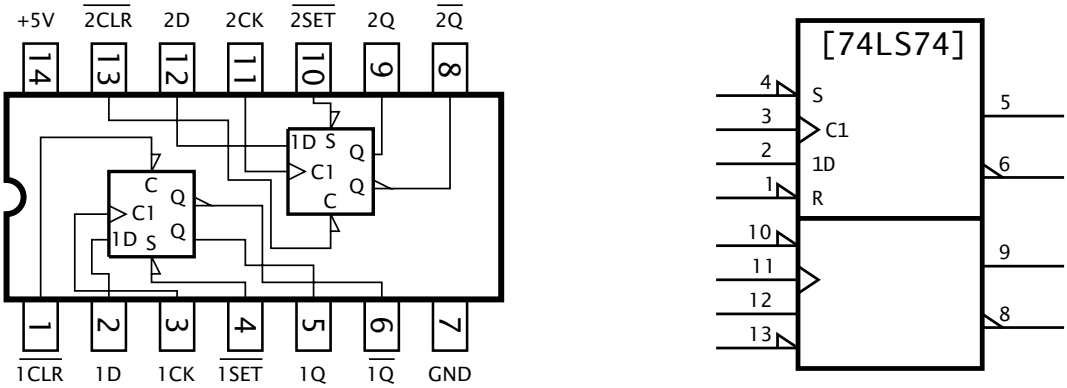
In Fig. 2.15(b) the dependency of the Data input with its Control is shown by the symbology C1 and 1D. The 1 prefix to D shows that it depends on any signal with a 1 suffix, in this case the C input. That is C1 clocks in the 1D data.

A flip flop is also a 1-bit memory cell, but the datum is only sampled on an *edge* of the control (known here as the Clock) input. The **D flip flop** described in Fig. 2.15(c) is triggered on a _/‾ (as illustrated in the truth table as ↑), but ‾_ clocked flip flops are common. The edge-triggered activity is denoted as > on a logic diagram, as shown in Fig. 2.15(d).

The 74LS74 shown in Fig. 2.16 has two D flip flops in the one SSI circuit. Each flip flop has an overriding Reset ($\overline{R}$) and Set ($\overline{S}$) input, which are asynchronous – that is not controlled by the Clock input. MSI functions include arrays of four, six and eight flip flops all sampling simultaneously with a common Clock input.

The 74LS377 shown in Fig. 2.17 consists of eight D flip flops all clocked by the same single Clock input C, which is gated by input $\overline{G}$. Thus the 8-bit data 8D...1D is clocked in on the _/‾ of C if $\overline{G}$ is Low. In the ANSI/ISO logic diagram shown in Fig. 2.17(b), this dependency is indicated as G1→1C2→2D, which states that $\overline{G}$ enables the Clock input, which in turn acts on the Data inputs.

Arrays of D flip flops are known as **registers**; that is read/write memories that hold a single word. The 74LS377 is technically known as a parallel-in parallel-out (PIPO) register, as data is entered in parallel (that is all in one go) and is available to read at one go. D latch arrays are also available, such as the 74LS373 octal PIPO register shown in Fig. 2.18, in which the eight D flip flops are replaced by D latches. In addition the



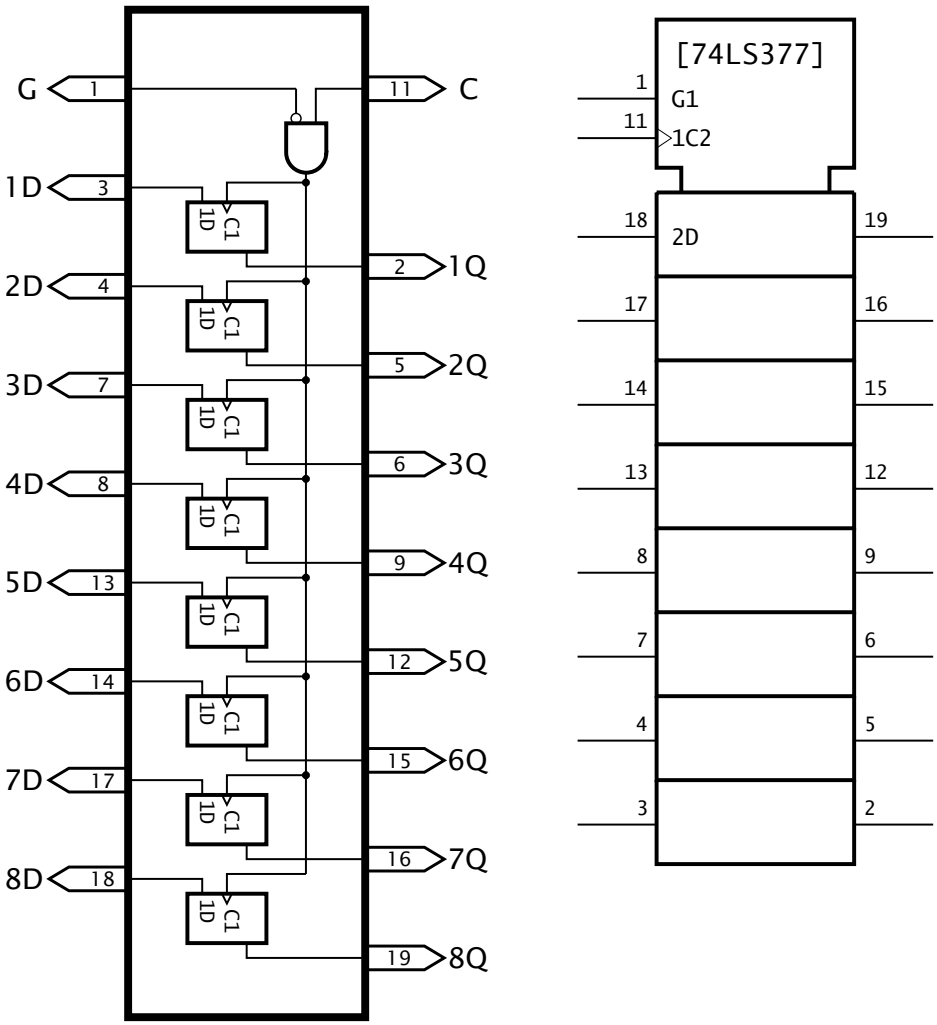
(a) Logic function *(b) ANSI/IEC logic symbol*

Fig. 2.16 The 74LS74 dual D flip flop.

latch outputs have a 3-state capability. This is useful if data is to be captured and later put onto a common data bus to be read subsequently as desired by a computer.

A pertinent example of the use of a PIPO register is shown in Fig. 2.19. Here an 8-bit ALU is coupled with an 8-bit PIPO register, accepting as its input the ALU output, and in turn feeding one input word back to the ALU. This register accumulates the outcome of a series of operations, and is sometimes called an **Accumulator** or **Working register**. To describe the operation of this circuit, consider the problem of adding two words A and B. The sequence of operations, assuming the ALU is implemented by cascading two 74LS382s might be:

1. Program step.
   - Mode = 000 (Clear).
   - Pulsing Execute loads the ALU output (0000 0000) into the register.
   - Data out is zero (0000 0000).
2. Program step.
   - Fetch Word A down to the ALU input.
   - Mode = 011 (Add).
   - Pulse _/‾_ Execute to load the ALU output (Word A + zero) into the register.
   - Data out is Word A.
3. Program step.
   - Fetch Word B down to the ALU input.
   - Mode = 011 (Add).
   - _/‾_ Execute to load the ALU output (Word B + Word A) into the register.



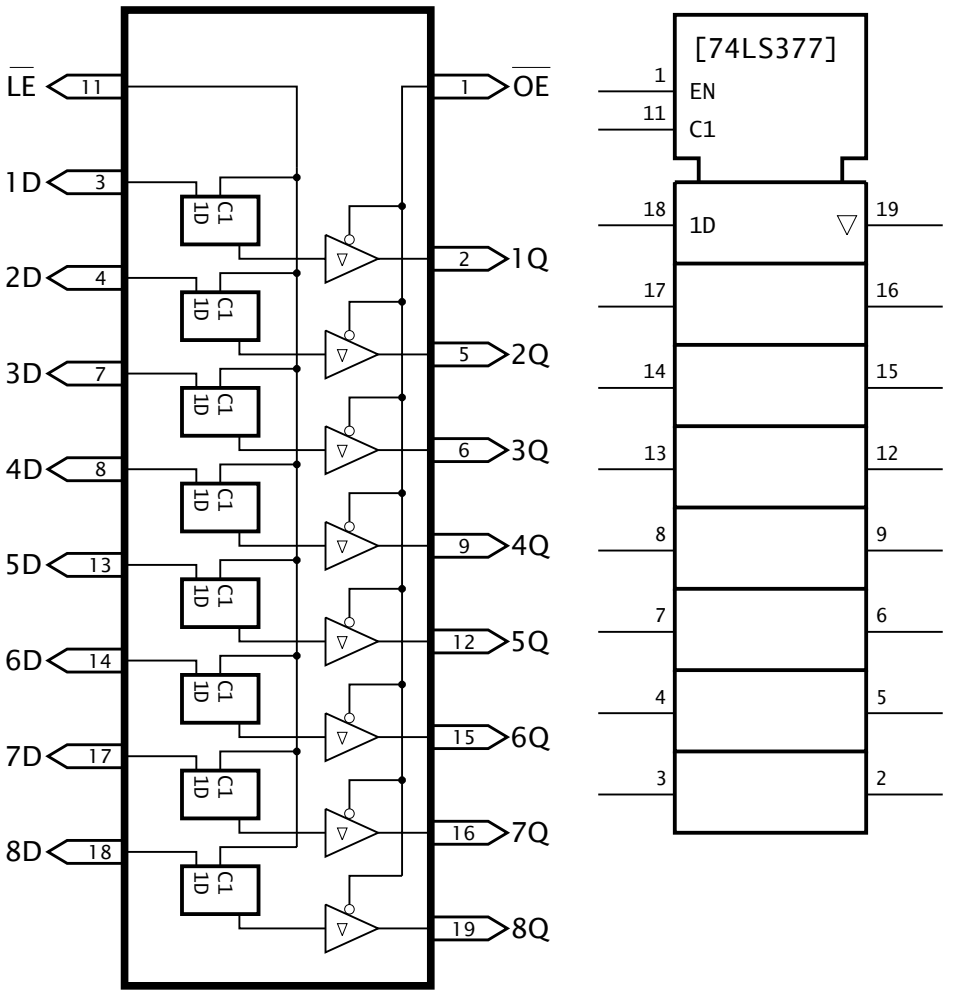
(a) Logic function *(b) ANSI/IEC logic symbol*

Fig. 2.17 The 74LS377 octal D flip flop array.

- Data out is Word B plus Word A.

The sequence of operation codes, that is 000 – 100 – 100 constitutes the program. In practice each instruction would also contain the address (where relevant) in memory of the data to be processed; in this case the locations of Word A and Word B.

Each outcome of a process will have associated properties. For example it may be zero or have a carry-out. Such properties may be significant in the future progress of the program. In the diagram three D flip flops, clocked by Execute, are used to grab this status information. In this situation the flip flops are usually known as **flags** (or sometimes semaphores). Thus we have **Z** (Zero), **C** (Carry from bit 7) and **DC** (BCD Carry from bit 3) flags, which form a Code Condition or Status register.



(a) Logic function *(b) ANSI/IEC logic symbol*

Fig. 2.18 The 74LS373 octal D latch array.

There are various other forms of register. The 4-bit **shift register** of Fig. 2.20(a) is an example of a serial-in serial-out (SISO) structure. In this instance the data held in the nth D flip flop is presented to the input of the $(n+1)$th stage. On receipt of a clock pulse (or shift pulse in this context), this data moves into this $(n+1)$th flip flop, i.e. effectively moving from stage n to stage $n+1$. As all flip flops are clocked simultaneously, the entire word moves once right on each shift pulse.

In the example of Fig. 2.20 a 4-bit external data nybble is fed into the left-most stage bit by bit as synchronised by the clock. After four shift pulses the serial 4-bit word is held in the register. To get it out again, four further shifts moves the wrod bit by bit out of the shift register, this is SISO. If the individual flip flops are accessible then the data can be accessed at one go, that is serial-in parallel-out.

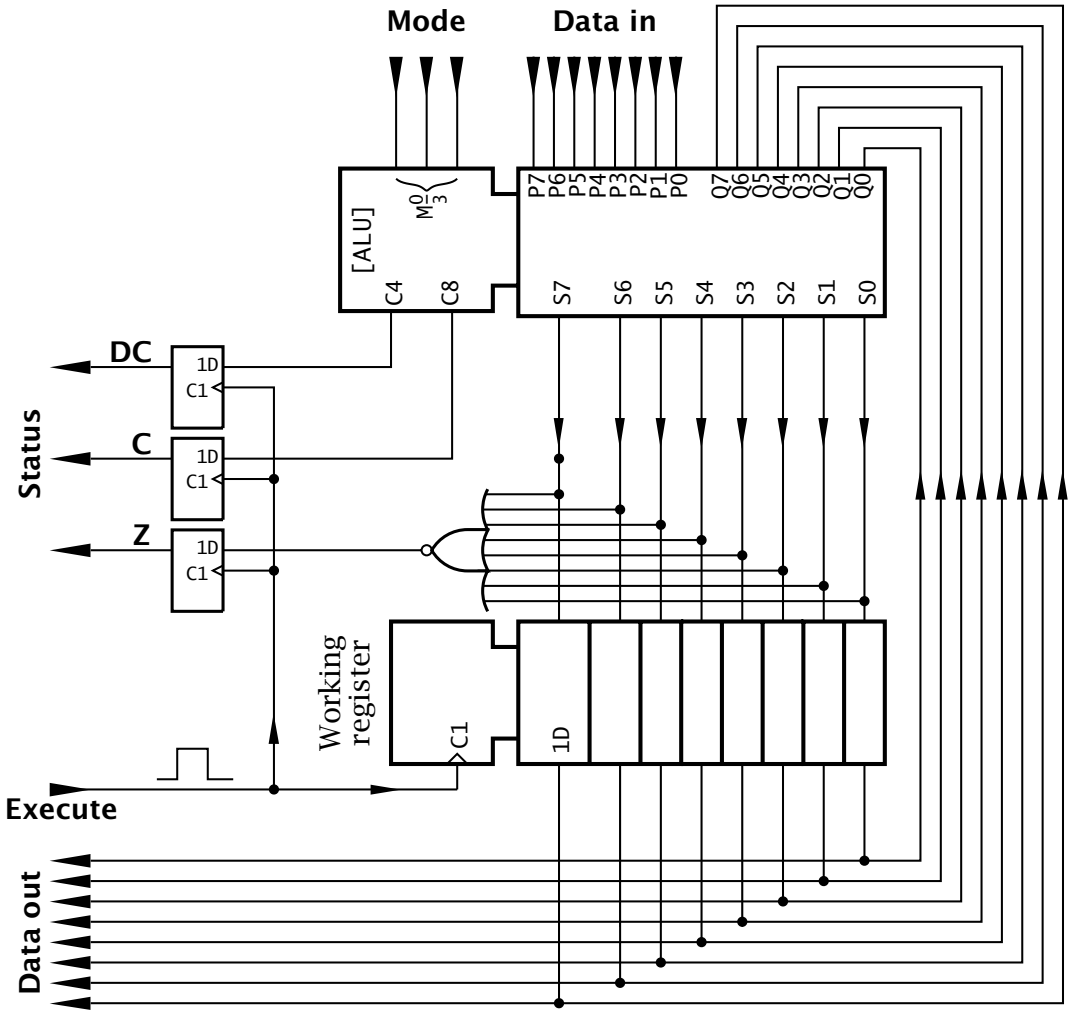
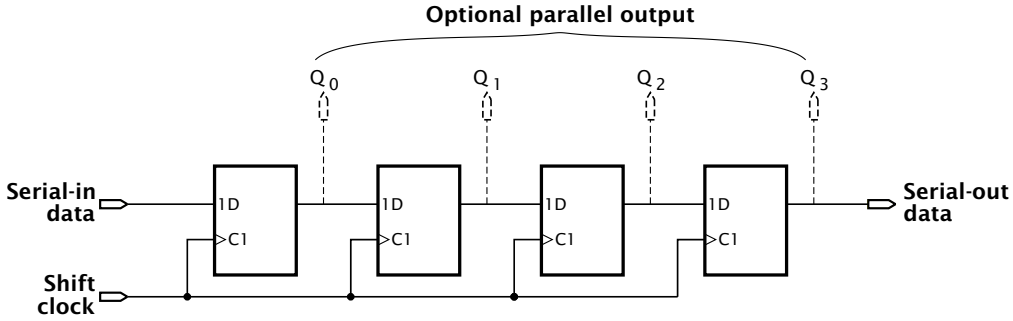


Fig. 2.19 An 8-bit ALU-accumulator processor.

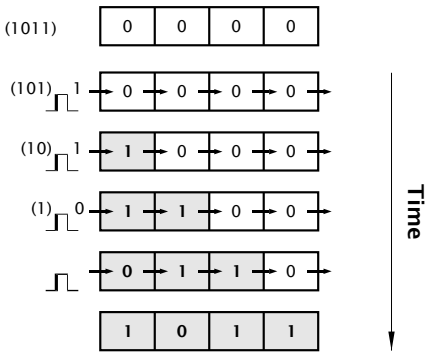
The logic diagram of Fig. 2.20(b) uses the → symbol affected by the clock input to indicate the shift action, C1 →. SRG4 indicates a Shift ReGister 4-stage architecture. An example of an 8-stage shift register is given in Fig. 12.2 on page 307.

Other architectures include parallel-in serial-out which is useful for parallel to serial conversion. Counting registers (counters) increment or decrement on each clock pulse, according to a binary sequence. Typically an n-bit counter can perform a count of 2^n states. Some can also be loaded in parallel and thus act as a store.

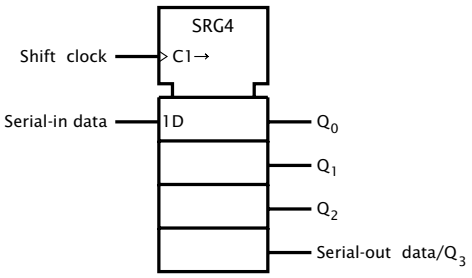
Consider the negative-edge triggered D flip flop shown in Fig. 2.21 where its $\overline{Q}$ output is connected back to the 1D input. On each ‾_ at the Clock input C1 the data at the 1D input will be latched in to appear at the Q output. As it is the complement of this output that is fed back to the input, then the next time the flip flop is clocked the *opposite* logic



(a) A 4-bit shift register



(b) Shifting 1011 into the register



(c) The ANSII/IEC logic symbol for a SIPO register

Fig. 2.20 The SISO shift register.

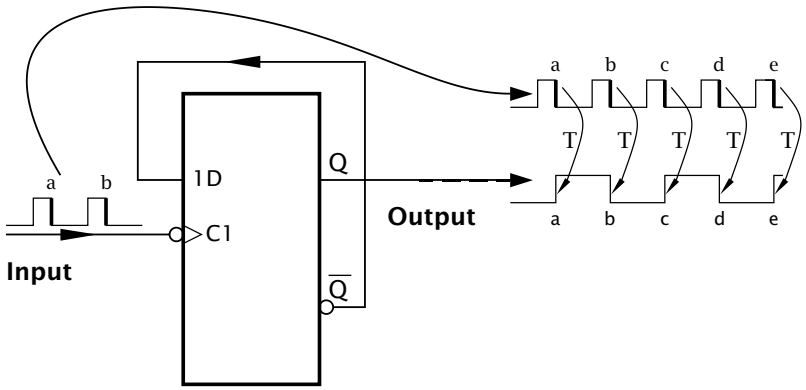
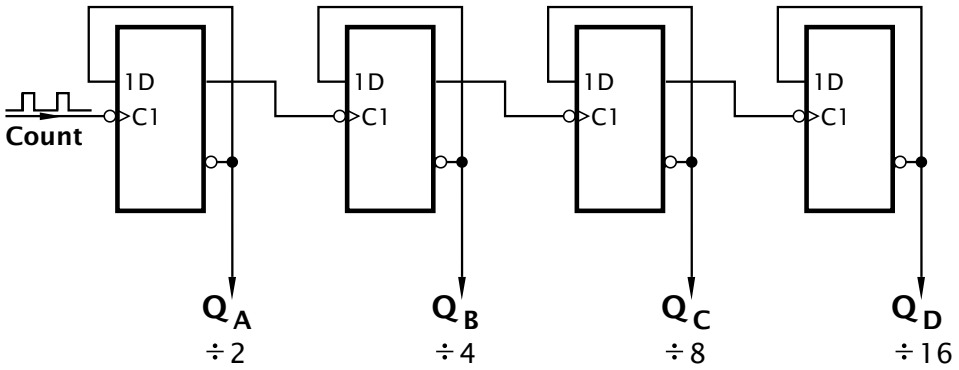


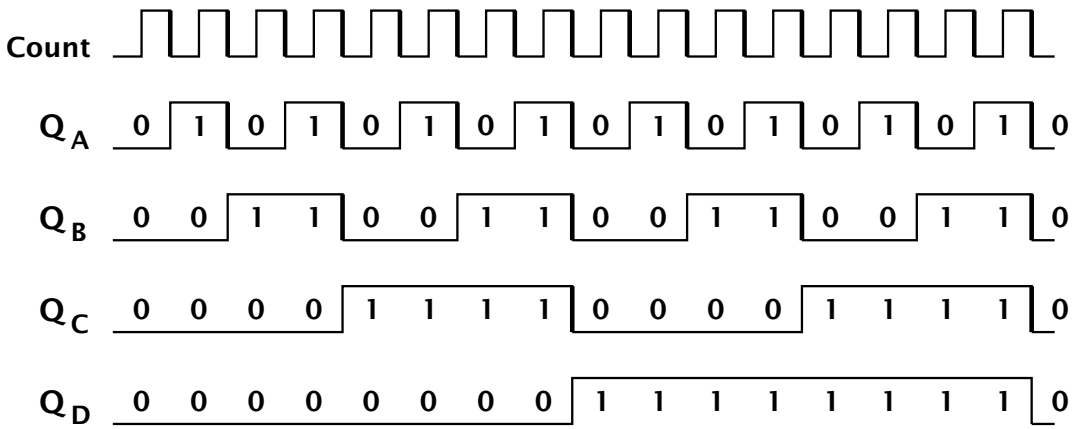
Fig. 2.21 The T flip flop.

state will be latched in. This constant alternation is called *toggling* and is depicted on the diagram by T. The output waveform resulting from a constant frequency input pulse train is half this frequency. This waveform is a precision squarewave, provided that the input frequency remains constant. This **T flip flop** is sometimes known as a binary or a divide-by-2.

T flip flops can of course be cascaded, as shown in Fig. 2.22(a). Here four ¯_ triggered flip flops are chained, with the output of binary n



(a) Cascading toggle flip flops



(b) Resulting waveforms

Fig. 2.22 A modulo-16 ripple counter.

clocking binary $n+1$. Thus if the input Count frequency was 8 KHz, then Q_A would be a 4 kHz square waveform and similarily Q_B would measure in at 2 kHz, Q_C at 1 kHz, Q_D at 500 Hz.

The waveform Q_A of Fig. 2.22(b) was derived in the same manner as in Fig. 2.21. Q_B is toggled on each $\overline{\ }\backslash\_$ of Q_A and likewise for the subsequent outputs. Marking a high as logic 1 and a low as logic 0 gives the 2^4 (16) positive-logic binary patterns as time advances, with the count rolling over back to state 0 on a continual basis. Each pattern remains in the register until the next event clocks the chain; an event being defined in our example as a $\overline{\ }\backslash\_$ at Count. Examining the sequence shows it to be a natural 8-4-2-1 binary up count, incrementing from $0000b$ to $1111b$. In fact the circuit is a modulo-16 **binary counter**. A modulo-n count is the sequence taking only the first n numbers into account.[8]

[8]Mathematically any number can be converted to its modulo-n equivalent by dividing by n. The remainder, or modulus, will be a number from 0 to $n-1$.

In theory there is no limit to the number of stages that can be cascaded. Thus using eight T flip flops would give a modulo-256 (2^8) counter. In practice there is a small propagation delay through each stage and this limits the ultimate frequency. For example the 74LS74 dual D flip flop of Fig. 2.16 has a maximum propagation from an event at its Clock input to output of 25 ns. The maximum toggling frequency for a single stage, such as in Fig. 2.21, is given as 25 MHz. An 8-stage counter thus has a maximum ripple-through time of 200 ns. If such a **ripple counter** were clocked at the resulting 5 MHz ($\frac{1}{200\,\text{ns}}$) then no sooner than one particular code pattern has stabilized then the next one would begin to appear. This is only really a problem if the various states of the counter are to be decoded and used to control other logic. The decoding logic, such as shown in Fig. 2.23, may inadvertently respond to these short transient states and cause havoc. In such cases more sophisticated synchronous counter configurations are more applicable where the flip flops are clocked simultaneously and steered by the appropriate logic configuration to count in the desired sequence.

The circuit illustrated here implements an up count. If the complement $\overline{Q}$ lines are used as the outputs, but with the clocking arrangements remaining the same, then the count sequence will decrement, that is a down count. Likewise using $\_/\overline{\ }$ triggered flip flops, such as the 74LS74 dual flip flop (see Fig. 2.23), are used as the storage element, then the count will be down. It is easily possible to use some simple logic to combine the two functions to produce a programmable up/down counter. It is also feasible to provide logic to load the flip flop array in parallel with any number and then count up or down from that point. Such an arrangement can be thought of as a parallel-in counting register.

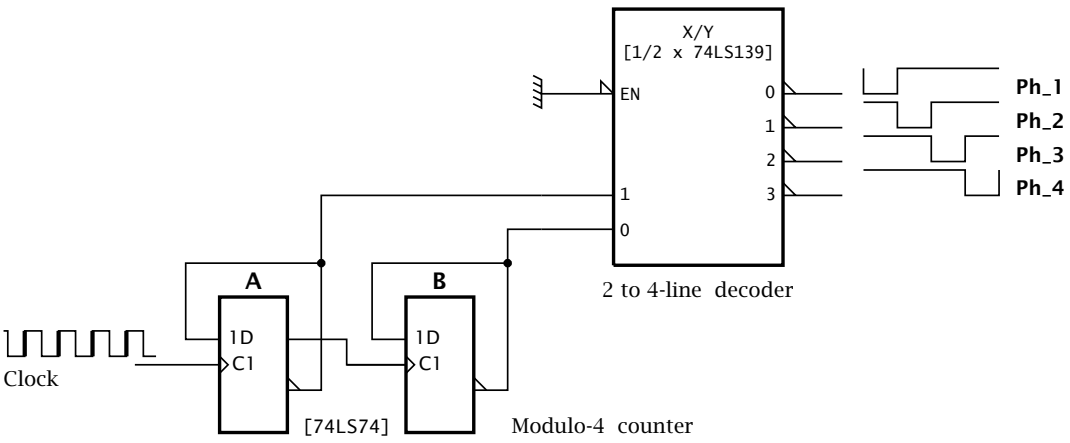
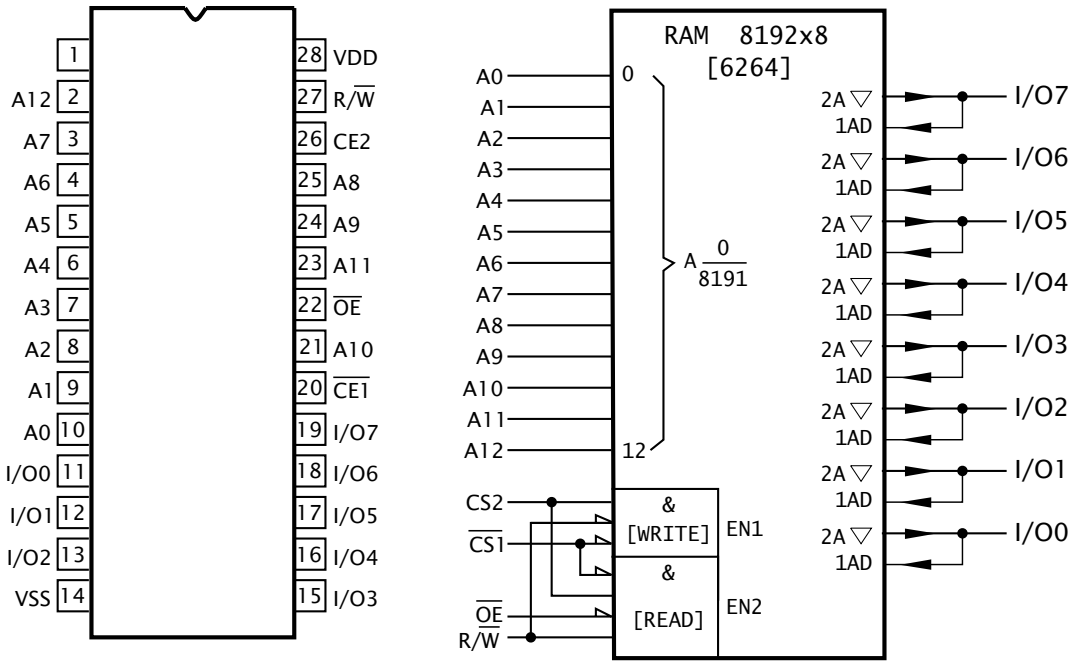


Fig. 2.23 Generating timing waveforms.

As well as the more obvious use of a counter register to totalize the number of events, such as cans of peas coming along a conveyor belt, there are other uses. One of these is to time a sequence of operations. In Fig. 2.23 a modulo-4 counter is used to address one section of a 74LS139 2 to 4-line decoder, see Fig. 2.5(a). This detects each of the four states of the counter, and the outcome is four time-separated outputs that can be used to sequence, say, the operation of a computer's control section logic. As a practical point, the complement $\overline{Q}$ flip flop outputs have been used to address the decoder to compensate for the $\_/\overline{\ }$ triggered action that would normally give a down count. Larger counters with the appropriate decoding circuitry can be used to generate fairly sophisticated sequences of control operations.

The term register is commonly applied to a read/write memory that can store a single binary word, typically 4 – 64 bits. Larger memories can be constructed by grouping n such registers and selecting one of n. Such a structure is sometimes known as a register file. For example, the 74LS670 is a 4×4 register file with a separate 4-bit data input and data output and separate 2-bit address. This means that any register can be read at any time, independently of any concurrent writing process.



(a) Dual in-line package *(b) Logic symbol*

Fig. 2.24 The 6264 8196 × 8 RAM.

Larger read/write memories are normally known as **read-write Random-Access Memories**, or **RAMs** for short. The term random-access indicates that any memory word may be selected with the same access time, irrespective of its position in the memory matrix.[9] This contrasts with a magnetic tape memory, where the reel must be wound to the sector in question – and if this is at the end of the tape!

For our example, Fig. 2.24 shows the 6264 RAM. This has a matrix of 65,536 (2^{16}) bistables organized as an array of 8192 (2^{13}) words of 8 bits. Word n is accessed by placing the binary pattern of n on the 13-bit Address pins A12...A0.

When in the Read mode (Read/$\overline{\text{Write}}$ = 1), word n will appear at the eight data outputs (I/O7...I/O0) as determined by the state n of the address bits. The A symbol at the input/outputs (as was the case in Fig. 2.11) indicates this addressibility. In order to enable the 3-state output buffers, the $\overline{\text{Output Enable}}$ input must be Low.

The addressed word is written into if R/$\overline{\text{W}}$ is Low. The data to be written into word n is applied by the outside controller to the eight I/O pins. This bi-directional traffic is a feature of computer buses.

In both cases, the RAM chip as a whole is enabled when $\overline{\text{CS1}}$ is Low and CS2 is High. Depending on the version of the 6264, this access from enabling takes around 100 – 150 ns. There is no upper limit to how long the data can be held, provided power is maintained. For this reason, the 6264 is described as static (SRAM). Rather than using a transistor pair bistable to implement each bit of storage, data can be stored as charge on the gate-source capacitance of a single field-effect transistor. Such charge leaks away in a few milliseconds, so needs refreshed on a regular basis. Dynamic RAMs (DRAMs) are cheaper to fabricate than SRAM equivalents and obtainable in larger capacities. They are usually found where very large memories are to be implemented, such as found in a personal computer. In such situations, the expense of refresh circuitry is more than amortized by the reduction in cost of the memory devices.

Both types of Read/Write memories are volatile, that is they do not retain their contents if power is removed. Some SRAMs can support existing data at a very low holding current and lower than normal power supply voltage. Thus a backup battery can be used in such circumstances to keep the contents intact for many months.

[9] Strictly speaking, ROMs should also be described as random access, but custom and practice has reserved the term for read-write memories.

CHAPTER 3

Stored Program Processing

If we take the Arithmetic Logic Unit (ALU)/Working register pair depicted in Fig. 2.19 on page 35 and feed it with function codes, then we have in essence a programmable processing unit. These command codes may be stored in digital memory and constitute the system's **program**. By *fetching* these **instructions** down one at a time we can execute this program. Memory can also hold data on which the ALU operates. This structure, together with its associated data paths, decoders and logic circuitry is known as a digital **computer**.

In Part 2 we will see that microcontroller architecture is modelled on that of the computer. As a prelude to this we will look at the architecture and operating rhythm of the computer structure and some characteristics of its programming. Although this computer is strictly hypothetical, it has been very much 'designed' with our book's target microcontroller in mind.

After reading this chapter you will:

- Understand the Harvard structure with its separate program and data memories, and how it compares to the more common von Neumann architecture.
- Understand the parallel fetch and execute rhythm and its interaction with the Program and Data stores and the internal processor registers.
- Understand the concept of a File address as a pointer to where data is located in the Data store.
- Comprehend the structure of an instruction and appreciate that the string of instructions necessary to implement the task is known as a program.
- Have an understanding of a basic instruction set, covering data movement, arithmetic, logic and skipping categories.
- Understand how Literal, Register Direct, File Direct, File Indirect and Absolute address modes permit an instruction to target an operand for processing.
- To be able to write short programs using a symbolic assembly-level language and appreciate its one-to-one relationship to machine code.

The architecture of the great majority of general-purpose computers and microprocessors is modelled after the **von Neumann** model shown in

Fig. 3.1.[1] The few electronic computers in use up to the late 1940s either only ever ran one program (like the war time code breaking Colossus) or else needed partly rewired to change their behavior (for example the ENIAC). The web site entry for this chapter gives historical and technical details of these prehistorical machines.

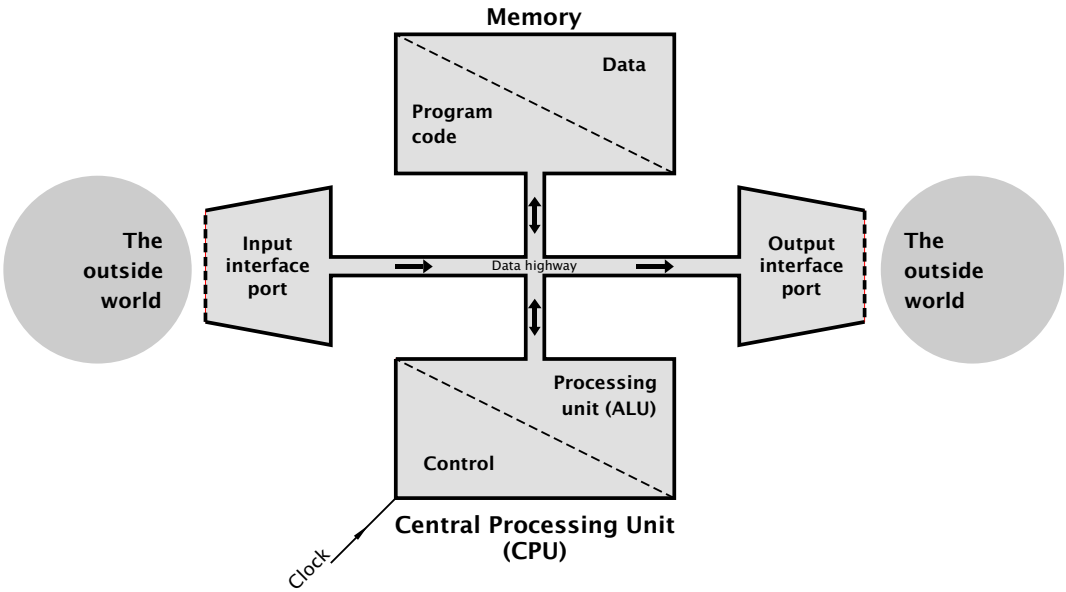


Fig. 3.1 An elementary von Neumann computer.

Von Neumann's great leap forward was to recognise that the program could be stored in memory along with any data. The advantage of this approach is flexibility. To alter the program simply load the bit pattern into the appropriate area of memory. In essence, the von Neumann architecture comprises a Central Processing Unit (CPU), a memory and a common connecting highway carrying data back and forth. In practice the CPU must also communicate with the environment outside the computer. For this purpose data to and from suitable interface ports are also funnelled through the data highway.

Looking at these elements in a little more detail.

[1] Von Neumann was a Hungarian mathematician working for the American Manhattan nuclear weapons program during the 2nd World war. After the war he became a consultant for the Moore School of Electrical Engineering at the University of Pennsylvania's EDVAC computer project, for which he was to employ his new concept where the program was to be stored in memory along with its data. He published his ideas in 1946 and EDVAC became operational in 1951. Ironically, a somewhat lower key project at Manchester University made use of this approach and the Mark 1 executed its first stored program in June 1948! This was closely followed by Cambridge University's EDSAC which ran its program in May 1949, almost two years ahead of EDVAC.

The Central Processing Unit

The CPU consists of the ALU/working register together with the associated control logic. Under the management of the control unit, program instructions are fetched from memory, decoded and executed. Data resulting from, or used by, the program is also accessed from memory. This fetch and execute cycle constitutes the operating rhythm of the computer and continues indefinitely, as long as the system is activated.

Memory

Memory holds the bit patterns which define the program. These sequences of instructions are known as the **software**. The word is a play on the term hardware; as such patterns do not correspond to any physical rearrangement of the circuitry. Memory holding software should ideally be as fast as the CPU, and normally uses semiconductor technologies, such as that described in the last chapter.[2] This memory also holds data being processed by the program.

Program memories appear as an array of cells, each holding a bit pattern. As each cell ultimately feeds the *single* data highway, a decoding network is necessary to select only *one* cell at a time for interrogation. The computer must target its intended cell for connection by driving this decoder with the appropriate code or **address**. Thus if location 602E*h* is to be read, then the pattern $\overset{6}{0}\overset{0}{0}\overset{2}{0}\overset{E}{1}10\ 0000\ 0010\ 1110b$ must be presented to the decoder. For simplicity, this address highway is not shown in Figs. 3.1 and 3.2.

This addressing technique is known as random access, as it takes the same time to access a cell regardless of where it is situated in memory. Most computers have large backup memories, usually magnetic or optical disk-based or magnetic tape, in which case access does depend on the cell's physical position. Apart from this sequential access problem, such media are normally too slow to act as the main memory and are used for backup storage of large arrays of data (eg. student exam records) or programs that must be loaded into main memory before execution.

The Interface Ports

To be of any use, a computer must be able to interact with its environment. Although conventionally one thinks of a keyboard and screen, any of a range of physical devices may be read and controlled. Thus the flow of fuel injected into a cylinder together with engine speed may be used to alter the instant of spark ignition in the combustion chamber of a gas/petrol engine.

---

[2] This wasn't always so; the earliest practical large high-speed program memories used miniature ferrite cores (donuts) that could be magnetized in any one of two directions. Core memories were in use from the 1950s to the early 1970s, and program memory is sometimes still referred to as core.

Data Highway
All the elements of the von Neumann computer are wired together with the one *common* data highway, or bus. With the CPU acting as the master controller, all information flow is back and forward along these shared wires. Although this is efficient, it does mean that only one thing can happen at any time. This phenomena is sometimes known as the von Neumann bottleneck.

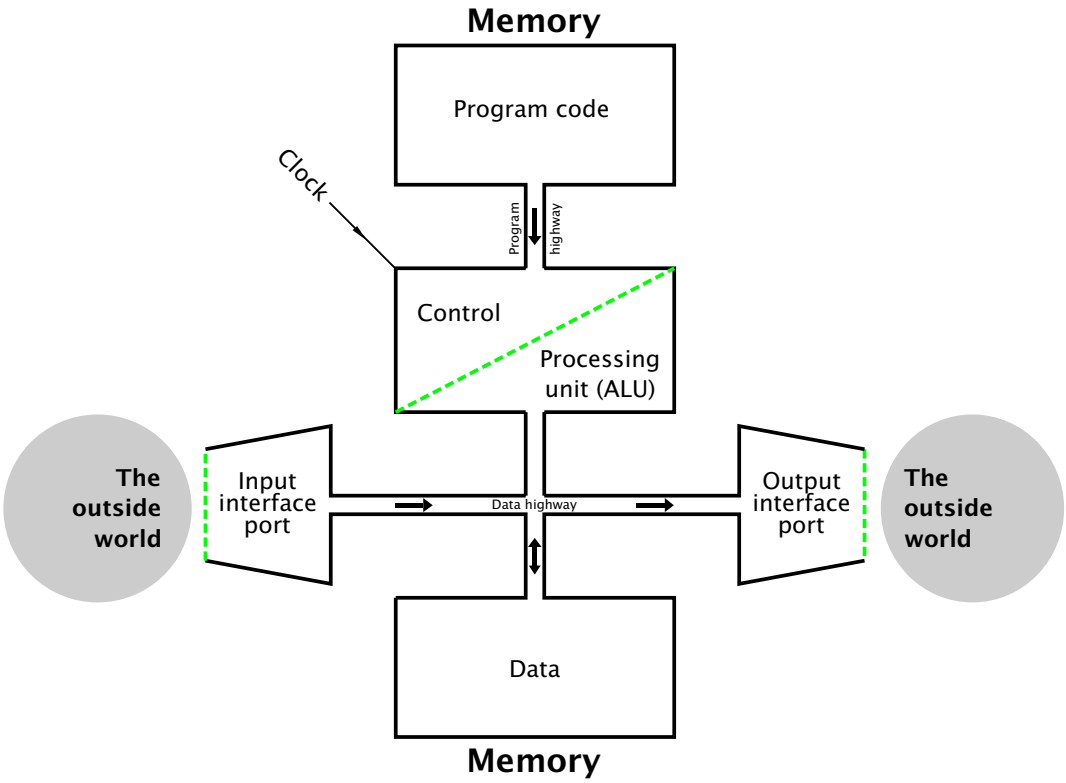


Fig. 3.2 An elementary Harvard architecture computer.

The **Harvard** architecture illustrated in Fig. 3.2 is an adaptation of the standard von Neumann structure, that separates the shared memory into entirely separate **Program and Data stores**. The diagram shows two physically distinct buses used to carry information to the CPU from these disjoint memories. Each memory has its own Address bus and thus there is no interaction between a Program cell and a Data cell's address. The two memories are said to lie in *separate memory spaces*. The Data store is sometimes known as the **File store**, with each location *n* being described as File n.

The fetch instruction down – decode it – execute sequence, the so called **fetch and execute cycle**, is fundamental to the understanding of

the operation of the computer. To illustrate this operating rhythm we look at a simple program that takes a variable called NUM_1, then adds $65h$ ($101d$) to it and finally assigns the resultant value to the variable called NUM_2. In the high-level language **C** this may be written as:[3]

```
NUM_2 = NUM_1 + 101;
```

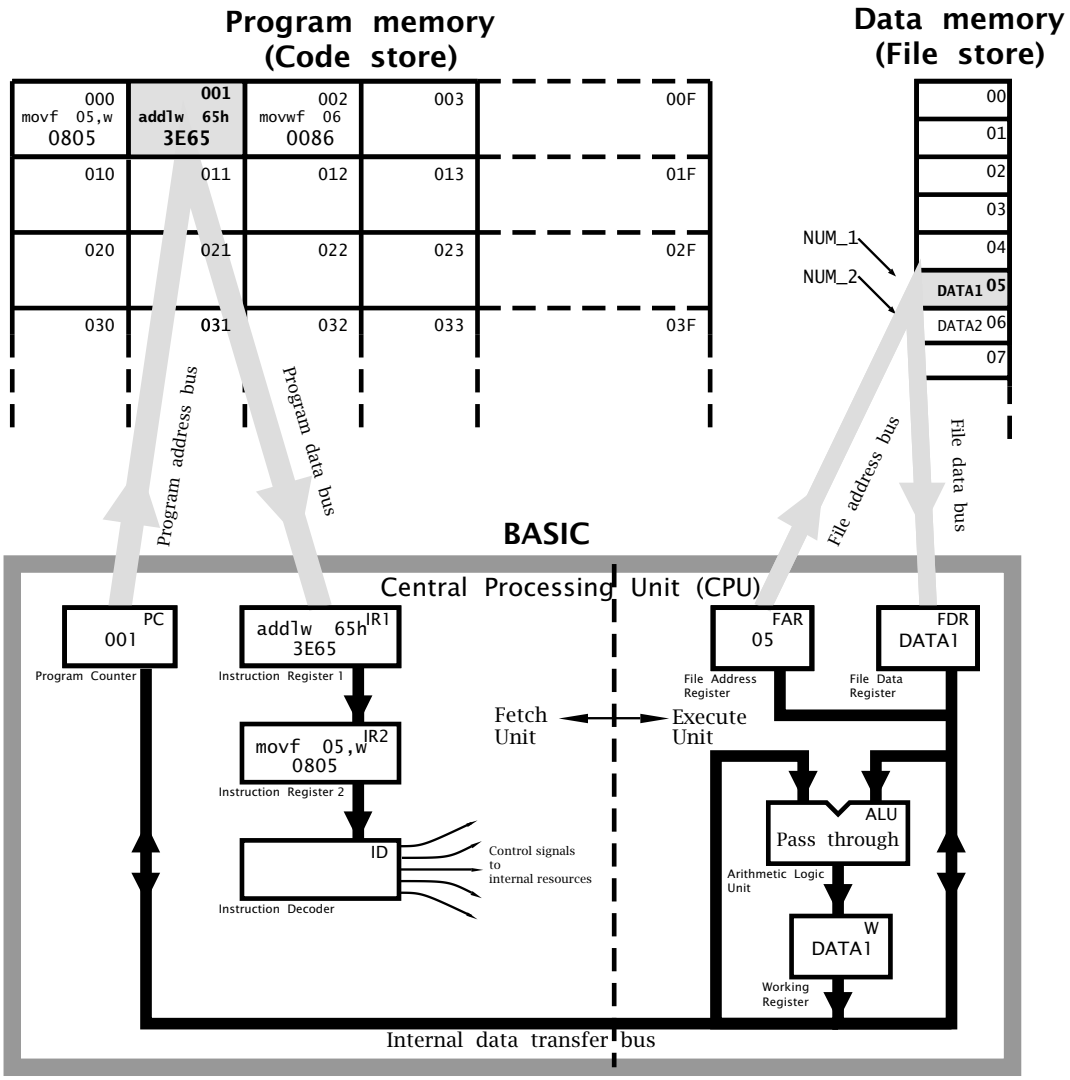


Fig. 3.3 A snapshot of the CPU executing the first instruction whilst fetching down the second instruction.

A rather more detailed close-up of our computer, which I have named BASIC (for Basic All-purpose Stored Instruction Computer) is shown in

[3] If you are more familiar with PASCAL or Modula-2, then this program statement would be expressed as `NUM_2 := NUM_1 + 101`

Fig. 3.3. This shows the CPU and memories, together with the two data highways (or **buses**) and corresponding address buses.

The CPU can broadly be partitioned into two sectors. The leftmost circuitry deals with *fetching* down the instruction codes and sequentially presenting them to the Instruction decoder. The rightmost sector *executes* each instruction, as controlled by this Instruction decoder.

Looking first at the fetch process:

Program Counter
Instructions are normally stored sequentially in Program memory, and the PC is the counter register that keeps track of the current instruction word. This up-counter (see Fig. 2.22 on page 37) is sometimes called (perhaps more sensibly) an Instruction Pointer.

As the PC is connected to the Execution unit – via the internal data bus – the ALU can be used to manipulate this register and disrupt the orderly execution sequence. In this way various Goto and Skip to another part of the program operations can be implemented.

Instruction Register 1
The contents of the Program store cell pointed to by the PC, that is instruction word n, is latched into IR1 and held for processing during the next cycle.

Instruction Register 2
During the same cycle as instruction word n is being fetched, the previously fetched instruction word $n-1$ in IR1 is moved into IR2 and feeds the Instruction decoder.

Instruction Decoder
The ID is the 'brains' of the CPU, deciphering the instruction word in IR2 and sending out the appropriate sequence of signals to the execution unit as necessary to locate the operand in the Data store (if any) and to configure the ALU to its appropriate mode. In the diagram the instruction shown is `movf 5,w` (MOVe File 5 to the Working register).

The Execution sector deals with accesses to the Data store and configuring the ALU. Execution circuitry is controlled from the Instruction Decoder, which is in turn commanded by Instruction word $n-1$ in IR2.

File Address Register
When the CPU wishes to access a cell (or file) in the Data store, it places the file address in the FAR. This directly addresses the memory via the File address bus. As shown in the diagram, File 5 is being read from the Data store and the resulting datum is latched into the CPU's File Data Register.

File Data Register
This is a bi-directional register which either:

- Holds the contents of an addressed file if the CPU is executing a **Read cycle**. This is the case for instruction 1 (`movf 5,w`) that moves (reads) a datum from File 5 into the Working register.
- Holds the datum that a CPU wishes to send out (Write) to an addressed file. This **Write cycle** is implemented for instruction 3 (`movwf 6`) that moves (writes) out the contents of the Working register to File 6.

Arithmetic Logic Unit
The ALU carries out an arithmetic or logic operation as commanded by its function code (see Fig. 2.9 on page 25) as generated by the Instruction Decoder.

Working Register
W is the ALU's working register, generally holding one of an instruction's operands, either source or destination. For example, `subwf 20,w` subtracts the contents of the Working register from the contents of File 20 and places the difference back in W. Some computers call this a Data register or Accumulator register.

In our BASIC computer, each instruction word in the Program store is 14 bits long. Some of these bits code the operation, for example $000111b$ for Add and $000110b$ for Exclusive-OR. This portion of the Instruction word is called the **operation code** or **op-code** (see Chapter 5). The rest of the instruction word bits generally relate to where in the Data store the operand is or sometimes a literal (constant) operand, such as in `addlw 6` (ADD Literal 6 to W). For example the instruction word for SUBtract File from W (`subfw`) is structured as op-code d fffffff, where:

- The op-code for Subtract is $000010b$ or $02h$.
- d is the destination for the difference, with 0 for W and 1 for a file, as specified below.
- fffffff is the 7-bit address of the subtrahend file (and destination if d is 1), from $00h$ through to $7Fh$.

For example `subwf 20h,w` is coded as $\boxed{000010}\boxed{0}\boxed{0100000}b$ or $0220h$.

As the Program and Data stores are separate entities, their cell size need not be the same. In our case each file holds an 8-bit byte datum. In consequence both the ALU and Working register are also byte sized. Generally the ALU size defines the size of the computer, and so BASIC could be described as an 8-bit machine. Real computers range in size from one bit up to 64 bits. From the previous example we see that seven bits of the instruction code are reserved for the file address, and thus the Data store has a maximum capacity for direct access limited to 128 (2^7) 8-bit files.

The Program memory capacity is a function of the Program Counter. If this were 10 bits wide, then the Program store could directly hold 1024 (2^{10}) 14-bit instructions. OK. We have got our CPU with its Program and Data stores. Let us look at the program itself. There are three instructions

in our illustrative software, and as we have already observed the task is to copy the value of a byte-sized variable NUM_1 plus $101d$ ($65h$) into a variable called NUM_2. This is symbolized as:

```
NUM_2 = NUM_1 + 101;
```

We see from our diagram that the variable named NUM_1 is simply a symbolic representation for "the contents of File 5" (which is shown as DATA1), and similarly NUM_2 is a much prettier way of saying "the contents of File 6" (shown as DATA2).

Now as far as the computer is concerned, starting at location $000h$ our program is:

```
00100000000101
11111001100101
00000010000110
```

Unless you are a CPU this is not much fun![4]

Using hexadecimal[5] is a little better.

```
0805
3E65
0086
```

but is still instantly forgettable. Furthermore, the CPU still only understands binary, so you are likely to have to use a translator program running on, say a PC, to translate from hexadecimal to binary.

If you are going to use a computer as an aid to translate your program, known as **source code**, to binary machine code, known as **object code**, then it makes sense to go the whole hog and express the program symbolically. Here the various instructions are represented by mnemonics (eg. `clrf` for CLeaR File, `subwf` for SUBtract File from W) and variables' addresses are given names. Doing this our program becomes:

```
movf  NUM_1,w    ; Copy the variable NUM_1 to W
addlw 101        ; Add the literal constant 101 decimal to it
movwf NUM_2      ; Copy NUM_1+101 into NUM_2
```

where the text after a semicolon is comment, which makes the program easier to understand by the tame human programmer.

Chapter 8 is completely devoted to the process of translation from this **assembly-level** source code to machine-readable binary. Here it is only necessary to look at the general symbolic form of an instruction, which is:

```
instruction mnemonic <operand A>,<operand B>
```

[4] I know; I have programmed this way back in the primitive middle 1970s.

[5] Remember that we are only using hexadecimal notation as a human convenience. If you took an electron microscope and looked inside these cells you would only 'see' the binary patterns indicated.

A few instructions have no explicit operand, such as `return` (RETURN from subroutine) and `nop` (No OPeration); however, the majority have one, two or even three operands. For instance, the operation Clear on its own does not make sense. Clear what? Thus we need to say "clear the destination file"; for example `clrf 20h` to clear File $20h$. Here Operand A is the file address $20h$. This could be written as `(f) <- 00`, where the brackets mean "contents of" and `<-` means "becomes". This notation is called **register transfer language** (**rtl**).

Many instructions have two operands. Thus the instruction `incf f,d` takes a copy of the contents of the specified file plus one and places this in either the Working register (`d = w`) or back in the file itself (`d = f`). For example, `incf 20h,w` means "deposit the contents of File $20h$ plus one in the Working register". In rtl this is `W <- (f20) + 1`.

Three-operand instructions are common. For example, `addwf f,d` adds the W register's contents to the specified file's contents and deposits the result either in `W` or in the file itself. Thus `addwf 20h,f` means "add the contents of `W` to that of File $20h$ and put the outcome in File $20h$" or `(f20) <- W + (f20)`. Of course this is not a true 3-operand instruction as the destination must be one of the two source locations; that is `W` or File $20h$. It is more accurately described as a $2\frac{1}{2}$-operand instruction!

All three instructions in our exemplar program have two operands, of which the Working register is either the source or/and destination. Where an instruction has a *choice* of destination `d` – as in `movf f,d` – then this is indicated appropriately as Operand B. Thus in our program `movf 5,w`. Where the source or destination is fixed as the Working register then this is indicated in the mnemonic itself, as in `addlw` (ADD Literal to W) and `movwf` (MOVe W to File).

The first and last instructions specify an *absolute* file, which is an actual location in the Data store. In a large program it is easier for us humans to give these variables symbolic names, such as `NUM_1` for File 5 and `NUM_2` for File 6. Of course we must somewhere tell the **assembler** (the program that does the translation) that `NUM_1` and `NUM_2` equate to addresses File 5 and File 6 respectively.

The middle instruction `addlw 101` adds a *constant number* or **literal** (that is $101d$ or $65h$) to `W` rather than a variable in memory. This literal is actually stored as part of the instruction word bit pattern (see page 107). In rtl this instruction implements the function `W <- W + 65`. In some cases it may be desirable to give a literal a symbolic name.

In writing programs using assembly-level symbolic representation, it is important to remember that each instruction has a one to one correspondence to the underlying machine instructions and its binary code. In Chapter 9 we will see that high-level languages loose that 1:1 relationship.

The essence of computer operation is the rhythm of the **fetch and execute cycles**. Here each instruction is successively brought down from

the Program store (fetched), interpreted and then executed. As any execution memory access will be on the Data store and as each store has its own busses, then the fetch and execution processes can progress *in parallel*. Thus while instruction n is being fetched, instruction $n-1$ is being executed. In Fig. 3.3 the instruction codes for both the imminent and current instructions are held in the two Instruction registers IR1 and IR2 respectively. This structure is known as a **pipeline**, with instructions being fetched into one end and 'popping out' into the Instruction decoder at the other end. Figure 3.4 below shows the time line of our 3-instruction exemplar program, quantized in clock cycles. During each clock cycle, except for the first, both a fetch and an execution is proceeding simultaneously.

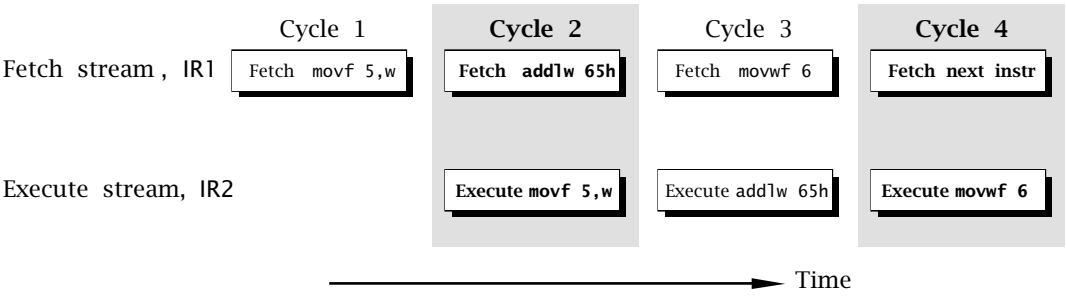


Fig. 3.4 Parallel fetch and execute streams.

In order to illustrate the sequence in a little more detail, let us trace through our specimen program. We assume that our computer (that is the Program Counter) has been reset to $000h$ and has just finished the Cycle 1 fetch.

Fetch (Fig. 3.3) ..... Cycle 2

- Increment the Program Counter to point to instruction 2.
- Simultaneously move the instruction word 1 down the pipeline (from Instruction register 1 to Instruction register 2).
- Program Counter ($001h$) to Program address bus.
- The instruction word 2 then appears on the Program data bus and is loaded into Instruction register 1.

Execute (Fig. 3.3) ..... Cycle 2

- The operand address $05h$ (i.e. NUM_1) to the File Address register and out onto the File address bus.
- The resulting datum at NUM_1 is read onto the File data bus and loaded into the File Data register.

- The ALU is configured to the Pass Through mode, which feeds the datum through to the Working register.

Fetch ..... Cycle 3

- Increment the Program Counter to point to instruction 3.
- Simultaneously move the instruction word 2 down the pipeline (from Instruction register 1 to Instruction register 2).
- Program Counter ($002h$) to Program address bus.
- The instruction word 3 then appears on the Program data bus and is loaded into the pipeline at Instruction register 1.

Execute ..... Cycle 3

- The ALU is configured to the Add mode and the literal (which is part of instruction word 2) is added to the datum in W.
- The ALU output, NUM_1 + $65h$, is placed in W.

Fetch ..... Cycle 4

- Increment the Program Counter to point to instruction 4.
- Simultaneously move instruction word 3 down the pipeline to IR2.
- Program Counter ($003h$) to Program address bus.
- The instruction word 4 then appears on the Program data bus and is loaded into the pipeline at IR1.

Execute ..... Cycle 4

- The operand address (i.e. NUM_2) $06h$ to the File Address register and out onto the File address bus.
- The ALU is configured to the Pass Through mode, which feeds the contents of W through to the File Data register and onto the File data bus.
- The datum in the File Data register is written into the Data store at the address on the File address bus and becomes the new datum in NUM_2.

Notice how the Program Counter is automatically advanced during each fetch cycle. This sequential advance will continue indefinitely unless an instruction to modify the PC occurs, such as `goto 200h`. This would place the address $200h$ into the PC, overwriting the normal incrementing process, and effectively causing the CPU to jump to whatever instruction was located at $200h$. Thereafter, the linear progression would continue.

Although our program doesn't do very much, it only takes around 1 μs to implement each instruction. A million unimpressive operations each second can amount to a great deal! Nevertheless, it hardly rates highly in

the annals of software, so we will wrap up our introduction to computing by looking at some slightly more sophisticated examples.

Writing a program is somewhat akin to building a house. Given a known range of building materials, the builder simply puts these together in the right order. Of course there are tremendous skills in all this; poor building techniques lead to houses that leak, are drafty and eventually may fall down!

It is possible to design a house at the same time as it is being built. Whilst this may be quite feasible for a log cabin, it is likely that the final result will not remain rain proof very long, nor will it be economical, maintainable, ergonomic or very pretty. It is rather better to employ an architect to design the edifice before building commences. Such a design is at an abstract level, although it is better if the designer is aware of the technical and economic properties of the available building materials.

Unfortunately much programming is of the 'on the hoof' variety, with little thought of any higher-level design. In the software arena this means devising strategies and designing data structures in memory. Again, it is better if the design algorithms keep in mind the materials of which the program will be built; in our case the machine instructions.

At the level of our examples in this chapter, it will be this coding (building) task we will be mostly concerned with. Later chapters will cover more advanced structures which will help this process, and we will get more practice at devising strategies and data structures.

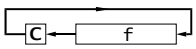
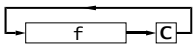
In order to code software we must have a knowledge of the register architecture of the computer/microcontroller and of the individual instructions. Figure 3.5 shows the **programming model** we will use for our exercises. This shows all registers that can be 'got at' by the programmer.

I have added three registers to the previous complement that actually are part of the Data store rather than the CPU but have a special significance for the programmer. File 0 and File 4 are a pair of Address registers working in tandem to point to an object in memory, as described further on in Fig. 3.6. The Status Register[6] **STATUS** comprises two bits in File 3 that are used to tell the software something about the outcome from an instruction. Thus the **C** flag (bit 0 in File 3) primarily holds the Carry out from the last Addition operation. For instance (in decimal) $5 + 2 = 7C0$; $5 + 9 = 4C1$. It also functions as the *complement* of the Borrow out from a Subtraction operation. For example $5 - 2 = 3\overline{B}1$; $5 - 9 = 6\overline{B}0$. The **Z** flag (bit 2 in File 3) is set if the result of the last operation is zero.

Table 3.1 shows all the instructions supported by the BASIC computer. Before looking at these, let us discuss the concept of the **address mode**. Most instructions act on data, which may be in internal CPU registers or out in the Data store. Thus the location of such operands must be part of the instruction. It isn't sufficient to simply state `clr` – Clear what? There

[6] Some processors use the term Code Condition register.

Table 3.1: Our BASIC computer's instruction set.

Instruction	Mnemonic	Operation	Flags	
			Z	C
Arithmetic				
Add W and F	`addwf f,d`	`(d) <- W + (f)`	√	√
Add Literal and W	`addlw L`	`W <- #L + W`	√	√
Clear F	`clrf f`	`(f) <- 00`	√	•
Clear W	`clrw`	`(f) <- 00`	√	•
Increment F	`incf f,d`	`(d) <- (f) + 1`	√	•
Decrement F	`decf f`	`(d) <- (f) - 1`	√	•
Subtract W from F	`subwf f,d`	`(d) <- (f) - W`	√	√
Subtract W from L	`sublw L`	`W <- #L - W`	√	√
Movement				
Move F	`movf f,d`	`(d) <- (f)`	√	•
Move W to F	`movwf f`	`W <- (f)`	•	•
Move Literal to W	`movlw L`	`W <- #L`	•	•
Logic				
AND W and F	`andwf f,d`	`(d) <- W · (f)`	√	•
AND Literal and W	`andlw L`	`W <- #L · W`	√	•
Complement F	`comf f`	`(f) <-` $\overline{(f)}$	√	•
Inclusive OR W and F	`iorwf f,d`	`(d) <- W + (f)`	√	•
Inclusive OR Literal and W	`iorlw L`	`W <- #L + W`	√	•
Rotate left F	`rlf f,d`	`(d) <-` 	•	√
Rotate right F	`rlr f,d`	`(d) <-` 	•	√
eXclusive OR W and F	`xorwf f,d`	`(d) <- W ⊕ (f)`	√	•
eXclusive OR Literal and W	`xorlw L`	`W <- #L ⊕ W`	√	•
Skip and Jump				
Bit Test F, Skip if Clear	`btfsc f,b`	`b == 0 ? PC++ : PC`	•	•
Bit Test F, Skip if Set	`btfss f,b`	`b == 1 ? PC++ : PC`	•	•
Decrement F, Skip if Zero	`decfsz f`	`--(f) == 0 ? PC++ : PC`	√	•
Go to address	`goto <ea>`	`PC <- <ea>`	•	•
Increment F, Skip if Zero	`incfsz f`	`++(f) == 0 ? PC++ : PC`	√	•

√ : Flag operates normally
• : Flag not affected
`<ea>` : Effective address
`b` : Bit `b` (0–7) in file
C : Carry or $\overline{\text{Borrow}}$, bit 0 in `F3`
Z : Zero, bit 2 in `F3`
`d` : Destination; 0 = `W`, 1 = file
`W` : Working register
`L` : 8-bit Literal
`==` : Equivalent to
`++` : Increment
`--` : Decrement
`A?S1:S2` : IF A is True THEN DO S1 ELSE DO S2

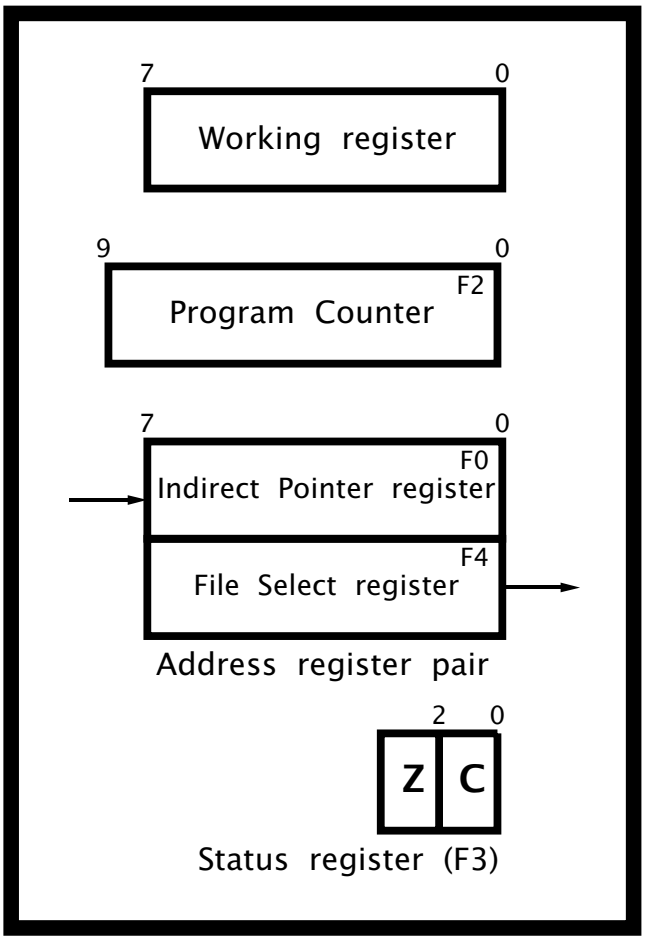


Fig. 3.5 Programmer's model.

are different ways of specifying the operand location. These are known as address modes.

Inherent
A few instructions, not shown in Table 3.1, do not explicitly refer to any location. For example `return` for RETURN from subroutine..

Register Direct
Here the operand is specified to be in the Working register. For example:

```
clrw        ; Clear the Working register
```

Most instructions specify at least one operand should be in W. In many cases W can be both a source and the destination operand. For instance:

```
addwf  f,w  ; Add W to a file and put the answer in W
```

where W initially holds one of the two source datum bytes, and ends up holding the outcome datum.

Literal
This address mode is used to specify an operand which is *constant* data rather than a location. For example:

```
addlw  120  ; Add the constant 120 (#120) decimal to W
```

Only special literal instructions are used with this type of data, such as `movlw` and `sublw`. The destination of this type of instruction is always `W`. The `sublw` instruction sometimes causes confusion as it actually takes away the contents of `W` from the literal byte and not vice versa. Thus `sublw 1` does not subtract one from the contents of `W` (i.e. decrement `W`) but $1 - W$. To decrement the contents of `W` use `addlw -1` (i.e. `addlw 0FFh`). Note the use of the # symbol in the rtl description to denote that the following number is *constant data*.

A few instructions can test or modify a single bit in a File. For instance:

```
btfsc 3Fh,6 ; Test File 3Fh bit 6; skip next instr. IF Clear
```

in which Operand B is the fixed literal 6, specifying the bit position in Operand A's specified File.

In both kinds of literal instruction the constant is encoded as part of the instruction word. In the former, eight bits of the 14-bit word is used, and in the latter three bits (see Appendix A).

File Direct
Here the totally fixed file address of the operand, either source or/and destination is directly specified. For example:

```
clrf 20h   ; Clear the byte at File store address 20h
```

clears the byte out at File 20*h*.

In many cases the same file can be both source and destination. Thus:

```
incf 20h,f ; Put the contents of File 20h plus 1 into File 20h
```

as opposed to the Working register:

```
incf 20h,w ; Put the contents of File 20h plus one into W
```

which uses File Direct addressing for Operand A and Register Direct for Operand B.

The main characteristic of this address mode is that the *location* of the operand is fixed as an integral part of the program, and thus cannot be changed as execution progresses. Although directly specifying its address may seem to be the obvious way to locate an object in the Data store, in some situations this technique is rather inflexible.

Suppose we wished to clear all file registers from 0C*h* through to 3F*h* in the File store, say to hold an array of 52 byte elements. The obvious

Program 3.1 Clearing a block of files the linear way.

```
CLEAR_ARRAY   clrf  0Ch ; Clear File 12
              clrf  0Dh ; and File 13
              clrf  0Eh ; Each clrf
              clrf  0Fh ; uses one instruction word
              clrf  10h ; in the Program store
              clrf  11h ; File 17 cleared
              clrf  12h ; and so on
              ....  ...
              ....  ...
              clrf  3Eh ; Clear File 62; nearly there
              clrf  3Fh ; Clear File 63; Phew!
```

way to do this is shown in Program 3.1, which uses a `clrf` instruction for each byte. Although it works, it is rather inefficient, and the mind boggles if you wanted to clear, say, a 1 Kbyte File store. There has got to be a better way!

File Indirect

Indirect addressing uses an address register to hold the address of an operand. This address register thus acts as a **pointer** into the Program store. The term **indirect** is used as this address register does not hold the operand datum itself, only a pointer address to it. The advantage of this seemingly perverse way of accessing data in the Data store, is that the **effective address** (**ea**) is a *variable* and this can be altered by the program as it progresses. This ea is the contents of this special pointer address register.

In our BASIC computer the **File Select Register** (**FSR**) is implemented as File 4 in the File store. The indirect mechanism is evoked when the dummy address File 0 (there is no physical file at location 0 in the Data store) is used for the operand address, as shown in Fig. 3.6.[7] Thus, for example, if the contents of File 4 happened to be $20h$ then:

```
movwf 0  ; Store datum in W out to place pointed to by File 4
```

will effectively copy the contents of W out to File $20h$.

This seems rather an obscure way of doing things, but by way of a justification let us revisit our array clearing example of Program 3.1. Repeating the same thing 52 times on successive file locations is a dubious way of doing this. Why not use a pointer into the array, and increment this pointer each time we do a Clear? That is, rather than using a constant address for the destination operand use a *variable* effective address.

Program 3.2 follows the scheme by folding the linear structure of the previous program into a **loop**, shown shaded. The execution path keeps

[7] Although this all seems rather complicated, I have kept in mind the microcontroller that we will be examining in the following chapters.

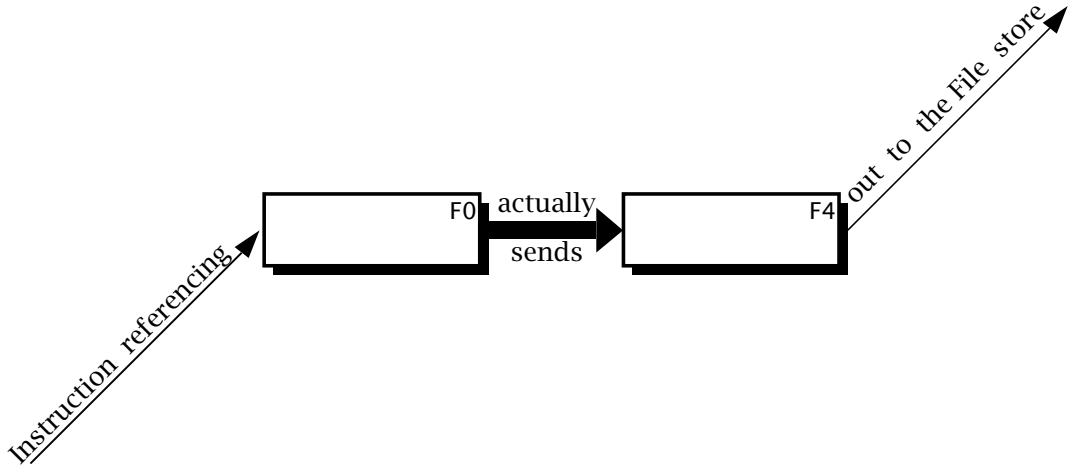


Fig. 3.6 The indirect mechanism.

circulating around the `clrf` instruction, which is 'walked' through the array of files by advancing the pointer in File 4, the FSR, on each pass through the loop. Eventually the pointer moves beyond the desired range and the program then exits the loop and continues onto the next section of code.

Program 3.2 has many new features, especially as we haven't yet reviewed the instruction set.

Program 3.2 Clearing a block of files using a repeating loop.

```
; Name  the  various  registers & bits  for  readibility
FSR  equ  4    ; Give File 4 the name FSR (File Select Reg.)
SR   equ  3    ; Give File 3 the name SR (Status Register)
Z    equ  2    ; The Z flag is bit 2 of the SR

; Now  for  the  program  proper
CLEAR_ARRAY movlw 0Ch    ; Put start address in W
            movwf FSR    ; and into the FSR as a pointer

CLOOP       clrf  0      ; Clear byte pointed to by FSR
            incf  FSR,f  ; Increment pointer in FSR
; Now  check,  is  pointer  at  top  yet?
            movf  FSR,w  ; Copy the pointer address into W
            sublw 40h    ; Compare with the end address (40h)
            btfss SR,Z   ; IF Zero flag in SR is set THEN fini
            goto  CLOOP  ; ELSE do the next pass thru the loop

            ....  ....   ; Next part of the program
```

Go here if FSR is not equal to 30h

Leave if FSR = 30h

Phase 1
From the point of view of readability, variables, registers and individual status and control bits should be given a relevant name. For example, `btfss STATUS,Z` (Bit Test File Skip if Set) which checks the **Z** flag (bit 2 in File 3) and skips the following instruction if set (that is the outcome of the previous instruction is zero) can be written in two ways; both of which are functionally identical:

```
btfss  3,2      ; IF bit 2 of File 3 is set THEN finished
btfss  STATUS,Z ; IF Zero flag in STATUS is set THEN finished
```

Obviously the latter is preferable. Although this might seem to be a cosmetic exercise, clarity reduces the chance of error and makes debugging and subsequent alteration easier. Realistic programs, rather than the code fragment illustrated here, use many variables and register bits, so lucidity is all the more important.

The three header lines of our program illustrate the means whereby the programmer tells the assembler translator program to substitute numbers for names. For example the line:

```
FSR  equ  4
```

states that when the programmer uses the name FSR as an operand, it is to be substituted by the number 4 (that is File 4). The `equ` directive means "equivalent to". A **directive** is a pseudo instruction in that it does not usually produce actual machine code but rather is a means of passing information from the programmer to the assembler program.

Phase 2
The first two proper instructions initialize the File Select Register to point to the first byte to be cleared, by moving the constant 0C*h* into the Working register and then out to File 4. Nearly all loop instructions involve some setting up before entry.

Phase 3
The key Clear instruction uses the File Indirect address mode by specifying the phantom File 0 as the destination address. This line has a **label** associated with it called CLOOP. The assembler knows that this is a label and not an instruction as it appears in the leftmost column of the source file. Lines without labels should begin with an indent of at least one space.

Each pass around the loop involves an incrementation of the pointer. This is done by using the `incf FSR,f` instruction to increment the FSR file. Notice that the destination here is specified as a file and not the W register.

Phase 4
Unless you wish to go round the loop forever, we need a mechanism to eventually exit. In our case this is done by comparing the contents of the

FSR pointer file with the constant 40*h*, that is with one over the top target file 3F*h*. The comparison mechanism is to copy the contents of the FSR pointer into W and then *subtract* W from the literal 40*h* using sublw 40h. If they are equal then the **Z** flag will be set and in this event the next btfss instruction will skip over the following goto LOOP instruction, out of the loop. Until this happens, the goto instruction will move the execution back up to the beginning of the loop and the process is repeated with the FSR advanced to point to the next file to be cleared.

All together our loop version takes eight machine instructions against the 52 of the linear equivalent. However, it does take longer to execute as the six instructions in the loop are repeated 52 times!

Absolute
The goto instruction forces the execution to directly transfer to the specified address by simply overwriting the Program Counter with the effective address – for example goto 145h puts 145*h* into the PC. It contrasts with the various Skip instructions (see page 61) which simply branch over the following instruction, wherever they are encountered. This latter type of branch is called relative, as opposed to the absolute transfer of execution used by the goto instruction.

Having covered the address modes, let us look briefly at the instruction set in Table 3.1. Instructions have been divided into four groups as follows.

Arithmetic
This covers the Addition, Subtraction, Incrementation, Decrementation and Clearing operations. Addition is possible between data in W and out in the File store, with the sum being placed either in the same file or in W. Similarly subtraction of W *from* a file is implemented, with the difference being placed in the same file or W. Both instructions come in a literal version, where W is added to/subtracted from an 8-bit constant, with the outcome remaining in the Working register. Notice that in the latter case the variable in W is *subtracted from* the literal rather than the more obvious subtraction of the literal from W.

The contents of any file can be incremented or decremented using incf and decf respectively. The outcome is usually put back into the file but the Working register can alternatively be specified as the destination.

Any file can be cleared with clrf, for instance clrf 20h. In the same manner the Working register can be zeroed with clrw.

Movement
This category of instructions *copy* a datum from source to destination. In all cases the contents of the source remains unaltered. movf reads (loads) a byte from the File store, usually into the Working register; eg. movf 20h,w. It is possible to specify the seemingly useless operation of copying a file's datum on top of itself, such as in move 20h,f. However,

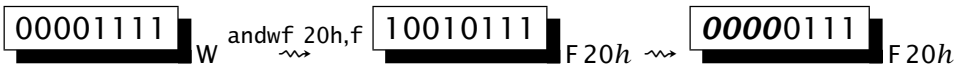
this does have the side effect of setting the **Z** flag if the datum is zero but in any case not altering a file's contents. This can be used to implement a Test for Zero operation.

Other instructions in this category write (store) the contents of W out into the File store (eg. `movwf 20h` copies W into File $20h$) or a literal into W (eg. `movlw 6` puts the constant $06h$ into W).

Logic

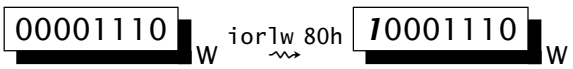
The `comf` instruction inverts (complements or NOTs) all bits in the specified file (see Fig. 1.1 on page 12). For example:

The `andwf` instruction bitwize ANDs (see Fig. 1.2 on page 13) the contents of the Working register to that of a specified file, with the outcome being either in W or in the file. For example:



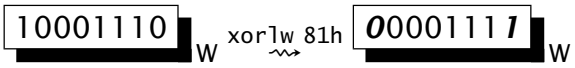
It is also possible to AND a constant to W by using the `andlw` variant.

Whilst the AND operation can be used to zero (mask) any bit or bits in the specified file, the Inclusive-OR equivalent instructions can set to one any bit or bits in the destination. For example, ORing W with the literal $10000000b$ ($80h$):

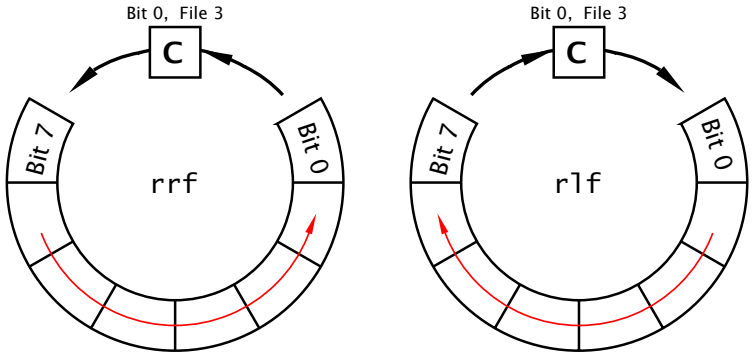


remembering from Fig. 1.3 on page 13 that ORing with a logic 1 always results in a logic 1 outcome and ORing with zero does not alter the original data, leads to the outcome that the most significant bit position of W is set and the other bits remain unaltered.

The `xorwf` and `xorlw` instructions provide for the eXclusive-OR operation. You will recall from page 14 that XORing with a 0 leaves a data bit unchanged, whilst XORing with a 1 inverts (or toggles) that bit. Thus, for example if we wished to invert both bits 0 and 7 of W:



Two instructions are provided that can shift the contents of any file once left or right, with the outcome either remaining in the file or appearing in the W register. As shown in Fig. 3.7 the outgoing bit (bit 7 for `rlf f,d` and bit 0 for `rrf f,d`) is placed in the **C** flag, whilst the incoming bit at the opposite end is the previous value of **C**. Because of this 'circular' action these instructions are said to implement a Rotate data through the Carry operation.



(a) Rotate Right File *(b) Rotate Left File*

Fig. 3.7 Circular shifts.

Skip and Jump

These instructions alter the state of the Program Counter, effectively interrupting the progressive flow of the program and causing processing to branch to another point in the code. The simplest of these instructions is `goto`. This overwrites the PC with the absolute destination address. For instance, `goto 200h` will change the PC to 200*h* and cause the program flow to 'jump' to whatever instruction is located at this address. Another example is instruction 8 in Program 3.2 where execution jumps back up to the beginning of the loop at address CLOOP.

The `goto` instruction causes an *unconditional* absolute jump in the program flow, implementing a 'Jump Always' operation. The remaining four instructions are *conditional* in that the smooth program progression is interrupted only if the outcome matches the specified state. For instance `decfsz` decrements the specified file contents and if and only if the outcome is zero the following instruction is skipped over.

Four instructions in our set come into this category, in that they increment the PC if some condition is fulfilled. Remembering that each instruction occupies one word in Program memory, incrementing the PC is equivalent to *skipping over* the following instruction. This *relative* skip-over action is in contrast with the *absolute* always jump to the specified address action of the `goto` instruction.

The four relative instructions are:

`btfsc`

Bit Test File & Skip if Clear by-passes the next instruction if the specified bit in the file in question is zero. For example `btfsc 3,02h` skips if bit 2 in File 3 is *clear*.

`btfss`

Bit Test File & Skip if Set by-passes the next instruction if the specified bit in the file in question is one. For example `btfss 3,02h` skips if bit 2 in File 3 is *one*.

decfsz
DECrement File & Skip if outcome is Zero subtracts one from the specified file and by-passes the next instruction if the outcome is zero. For example `decfsz 30h,f` skips if the content of 30*h* once decremented is *zero*. The decremented value is placed back in the file in this example but W is an alternative destination, as in `decfsz 30h,w`.

incfsz
INCrement File & Skip if outcome is Zero adds one to the specified file and by-passes the next instruction if the outcome is zero. For example `incfsz 30h,f` skips if the content of 30*h* once incremented is *zero*. The incremented value is placed back in the file in this example but W is an alternative destination, as in `incfsz 30h,w`.

Actually the Program Counter is located in the Data store as File 2. Thus manipulating this file can implement a computed relative skip. For example:

```
movf  2,w  ; Bring the current value of the PC into W
addlw 6    ; Add six to it
movwf 2    ; Update PC, that is hop forward six places
```

More details of this technique are given on page 86.

As more sophisticated example of the use of conditional Skip instructions and the absolute `goto` instructions, consider the problem of repeating a sequence 16 times. The most efficient approach to the coding is to construct a program loop and keep track of the number of times the processor executes the encapsulated instructions. In the following listing File 22*h* is used as the counter, initialised to 16 *before* the loop is entered. At the end of the sequence the count is decremented using `decfsz` and normally the next instruction, which transfers the PC back up to the start of the loop, is executed. However, eventually the count reaches zero and the `goto LOOP` instruction is skipped over, with execution consequently exiting the loop. Traditionally the potentially skipped over instruction is shown indented as a matter of style.

```
     movlw  16   ; Set up constant #16
     movwf  22h  ; and put in File 22h for use as a counter

; Now for the sequence of instructions to be repeated 16 times
LOOP .....  ...  ; DO this
     .....  ...  ; DO that
     .....  ...  ; DO the other

; Count mechanism
     decfsz 22h  ; Decrement count. IF zero THEN exit loop --
      goto  LOOP ; ELSE repeat loop                        |
                                                           |
     .....  ...                                          <--
```

The incfsz instruction works in a similar manner, but for an up count, skipping when the target file overflows FF → 00*h*. As an exercise, repeat the example but using incfsz in place of decfsz.The ++(f) and --(f) rtl symbology used to describe the incfsz and decfsz instructions indicate that the contents of the designated file is augmented or decremented *before* testing for zero.

The other set of skip instructions uses the state of *any bit* b in any file to force a conditional skip. Thus if it is desired that the program transfers to an instruction located at label REAL_TIME if bit 2 of File 0B*h* is logic 1, then we can use the btfsc (Bit Test File & Skip on Clear) instruction thus:

```
 btfsc  0Bh,2      ; Bit Test File 0Bh bit 2. IF==0 THEN skip
  goto  REAL_TIME  ; ELSE == 1, GOTO REAL_TIME
```

In Table 3.1 the rtl language description of this instruction is given as b==0?PC++:PC, which can be read as:

1. Is the specified bit (b) equivalent to (==) zero (?).
2. If true then increment the PC (PC++), that is skip.
3. Else it is false, so leave the PC alone.

Similarly the -- rtl operator is used to indicate decrementation.

Examples

Example 3.1

Write a program to add the byte *contents* in File memory called NUM1 (File 20*h*) to NUM2 (File 21*h*). The outcome is to be in SUM_H and SUM_L (File 22*h* and File 23*h* respectively) in the order high:low byte.

Solution

This is similar to our load-add-store program on page 48 but the second operand is a byte variable in memory rather than a constant. The three instructions to implement this are shown in Program 3.3. The variable NUM1 is simply fetched down into the Working register and then added to variable NUM2. The outcome of this is then copied into the sum byte SUM_L.

Of course this will only work if the outcome of the addition can fit into a single byte; that is no more than FF*h* (255*d*). If, for example, both NUM1 and NUM2 were FF*h*, then the outcome would be 1 FF*h*. Thus we need to reserve two bytes for the sum; an upper byte and a lower byte; named SUM_H and SUM_L in Program 3.4 in File 22*h* and File 23*h* respectively. In this implementation we simply zero the upper byte of the sum in advance, and after the addition *skip* around the Increment of instruction 6 if the

Program 3.3 Simple single-precision addition of two byte variables.

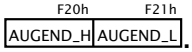
```
NUM1    equ   0020h
NUM2    equ   0021h
SUM_L   equ   0022h

SP_ADD  movf  NUM1,w  ; Get the first memory byte
        addwf NUM2,w  ; Add to it the second byte and put the
        movwf SUM_L   ; outcome in memory as the lower sum byte
```

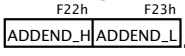
Program 3.4 A more accurate single-precision addition of two byte variables.

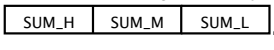
```
NUM1    equ   0020h
NUM2    equ   0021h
SUM_L   equ   0022h
SUM_H   equ   0023h
STATUS  equ   03

SP_ADD  clrf SUM_H     ; Prepare the upper sum byte by zeroing it

        movf  NUM1,w   ; Get the first memory byte
        addwf NUM2,w   ; Add to it the second byte and put the
        movwf SUM_L    ; outcome in memory as the lower sum byte

        btfsc STATUS,0 ; IF carry is clear (SR bit 0) THEN finish
        incf  SUM_H,f  ; ELSE increment higher sum byte

EXIT    ...  .....   ; Next part of the program
```

Carry flag (bit 0 in File register 3) is Clear (Bit Test File and Skip on Clear). The upper sum byte can only ever be either $00h$ or $01h$.

Example 3.2

Write a program routine that will add two 16-bit numbers giving a 17-bit sum. The augend is located in the two memory locations $F20h$:$F21h$ in the order high:low byte thus 

. The addend is similarly situated 

. The sum is stored as three bytes in the order high:middle:low thus 

.

Solution

Although BASIC is only capable of directly implementing 8-bit arithmetic, operations of any length are possible by breaking down the process into byte-sized chunks. In the case of addition, this involves a sequence of byte operations from the least to the most significant digits with any carry from the nth digit byte being added into the $n+1$th summation. The least significant addition has a presumed carry-in of 0 and the carry-out from

the most significant addition becomes the highest bit of the outcome. For example FF FFh + FF FFh = 1 FF FFh (65,535d + 65,635d = 131,070d).

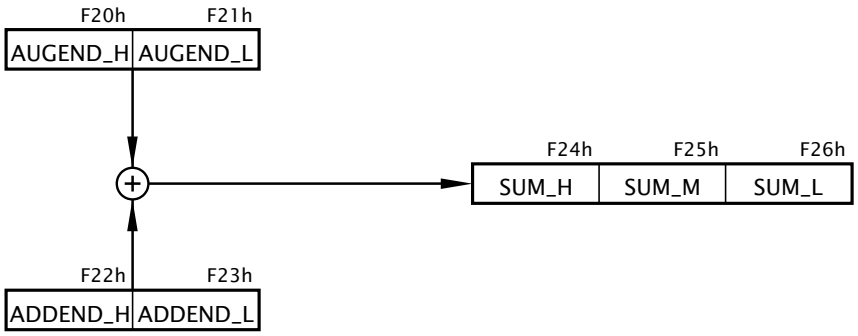
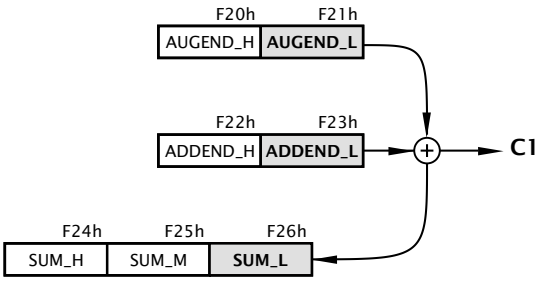


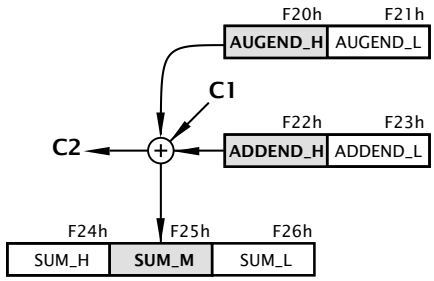
Fig. 3.8 The process.

The overall process is diagrammatically shown in Fig. 3.8. However, given that we need to implement the process as a sequence of steps executable by the byte-sized instructions of Table 3.1 then the next step is to produce a task listing.

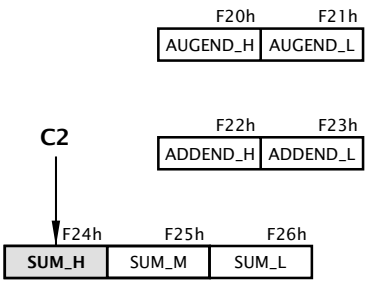
1. Add the low bytes of the augend and addend, generating the low byte of the sum and carry C1.



(a) Adding the least-significant bytes



(b) And the most-significant bytes



(c) The most-significant sum byte is the last carry-out

Fig. 3.9 Visualization of the task process.

2. Add the high bytes of the augend and addend plus the last carry-out C1 to give the middle byte of the sum and a new carry-out C2.
3. The high byte of the sum is the last carry-out C2, either 0 or 1.

Given that this is our first program of any substance, a detailed visualization of this task list will be useful. For most instances detail at this level is not helpful and subsequently we will use a more abstract visualization known as a **flow chart** (see Example 3.3).

Once a task list has been established then the next step is to implement this as a sequence of instructions, that is the program. One possible is shown in Program 3.5.

In the listing the three tasks are identified by an appropriate comment.

Task 1
This comprises a Load-Add-Store sequence to add the lower byte of the addend to that of the similarly significant augend. The outcome byte is

Program 3.5 The double-precision add program.

```
AUGEND_H    equ 020h
AUGEND_L    equ 021h
ADDEND_H    equ 022h
ADDEND_L    equ 023h
SUM_H       equ 024h
SUM_M       equ 025h
SUM_L       equ 026h
STATUS      equ   03      ; STATUS flag register
C           equ   0       ; The C flag

DP_ADD   clrf  SUM_M      ; Zero the 2 upper bytes of the sum
         clrf  SUM_H

; Task 1
         movf  AUGEND_L,w ; Get LSB of the augend
         addwf ADDEND_L,w ; Add LSB of addend
         movwf SUM_L      ; and put it away

; Tasks 2 and 3
         btfsc STATUS,C   ; Was there a carry C1?
          incf SUM_M      ; IF yes THEN add to middle sum byte

         movf  AUGEND_H,w ; Get MSB of the augend
         addwf ADDEND_H,w ; Now add to MSB of the addend
         btfsc STATUS,C   ; Was there a carry C2?
          incf SUM_H      ; IF so THEN add to high byte sum

         addwf SUM_M,f    ; Add the outcome to mid sum byte
         btfsc STATUS,C   ; Was there a carry C2 from this?
          incf  SUM_H     ; IF so THEN add to high sum byte

         ....  .....      ; Next program
```

stored in memory at File $26h$ (SUM_L) and the Carry flag bit (bit 0 in File 3) is set as appropriate to C_1.

Task 2
If there was a carry-out from Task 1 then the precleared middle byte of the sum is incremented as the carry-in C_1. Then the upper bytes of the addend and augend are added. The final outcome of SUM_M (in File $25h$) is determined by adding its current state ($00h$ or $01h$) to the outcome of this addition.

Task 3
Either of the two additions in Task 2 can result in a carry-out. The C flag in the STATUS register can be tested after each addition and if set, the upper byte of the sum SUM_H (File $26h$) is incremented to add C_2, as shown in Fig. 3.8(c). There will not be a carry-out generated by both the Task 2 additions.

Example 3.3

Write a program to divide the byte in the Working register by ten. The quotient is to be in File $20h$ and the remainder in W.

Solution

Division is the process in finding how many times the divisor (ten in our case) can be subtracted from the dividend without underflowing, that is producing a borrow.

The flow chart shown in Fig. 3.10 outlines the task list to implement our algorithm. Initializing the quotient to -1 means that the subtract and increment loop can begin by incrementing, and this process will continue until a borrow is produced after the subtraction by ten process. A

Program 3.6 Dividing by ten.

```
QUOTIENT equ  020h         ; Where the quotient will be stored
STATUS   equ  03           ; Where STATUS reg. flags are located

DIV_10   clrf  QUOTIENT    ; 1: Zero quotient
         decf  QUOTIENT,f  ; 1: Initial value is -1

DIV_LOOP incf  QUOTIENT,f ; 2: Increment quotient

         addlw -10         ; 3: Subtract ten from dividend

         btfsc STATUS,0    ; 4: Leave loop if borrow (bit 0 = 0)
          goto  DIV_LOOP   ; 4: ELSE repeat increment and subtract

         addlw 10          ; 5: Add ten to give remainder

         ....  .....       ; Next program
```

borrow-out indicates that the last subtraction was not successful in that the divisor (i.e. ten) was larger than the residue it was subtracted from. That is the outcome is negative. The loop count on exit represents the number of successful subtractions and thus the quotient.

The coding in Program 3.6 closely follows the flow chart, with comment numbers referring to the statement boxes. The actual subtract constant ten is implemented by adding minus ten! This is because the more obvious `sublw 10` instruction actually subtracts W from ten and not ten from W. Notice the mechanism for exiting the loop by testing and skipping if bit 0 in STATUS is clear after this subtraction.

Remembering that bit 0 in File 3 is the Carry flag doubling as the $\overline{\text{Borrow}}$ flag, then a borrow out is indicated if this bit is *zero* after the subtraction. On exit ten is added back on to compensate for the last unsuccessful subtraction and the outcome then gives the remainder.

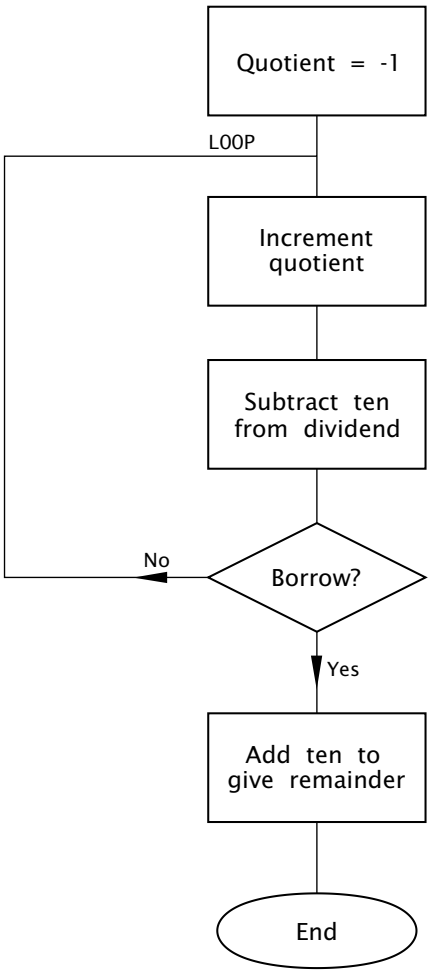


Fig. 3.10 Division by repetitive subtracting.

Example 3.4

As part of a program to convert degrees Celsius to Fahrenheit it is necessary to multiply the byte in W by nine. Devise a suitable coding.

Solution

There are two ways of multiplying. The fundamental definition of multiplication is repetitive addition, thus we could add the datum nine times to implement our specified function. An alternative approach is the shift and add technique outlined on page 11. Thus the $\times 9$ function is implemented as $\times 8 + \times 1$. The former is carried out by shifting left three times (i.e. 2^3). Thus we have:

$$W \times 9 = W \times 8 + W \times 1 = (W << 3) + W$$

No matter what technique is used, the outcome by definition will be larger than either the multiplier or multiplicand. In this case we need two bytes. In the coding of Program 3.7 the two memory locations File $20h$ and File $21h$ are used to hold the high byte and the low byte respectively of the final 2-byte product. By copying the multiplicand from W to the low byte and clearing the upper byte we make an extended 2-byte version of the multiplicand in Data memory. Shifting left is implemented by using

Program 3.7 Multiplying by nine.

```
STATUS     equ  3              ; STATUS register flags
C          equ  0              ; Carry flag is bit0
PRODUCT_H  equ  020h           ; High byte of the product
PRODUCT_L  equ  021h           ; Low byte

MUL_9      movwf  PRODUCT_L    ; Move the multiplicand to memory
           clrf   PRODUCT_H    ; Extend it to 2 bytes

; Now shift the double byte multplicand three times to give x8
           bcf    STATUS,C     ; Clear carry
           rlf    PRODUCT_L,f  ; Shift left low byte of product
           rlf    PRODUCT_H,f  ; and the high byte
           bcf    STATUS,C     ; Clear carry
           rlf    PRODUCT_L,f  ; Shift left low byte of product
           rlf    PRODUCT_H,f  ; and the high byte
           bcf    STATUS,C     ; Clear carry
           rlf    PRODUCT_L,f  ; Shift left low byte of product
           rlf    PRODUCT_H,f  ; and the high byte

; Now add the original value giving x8 + x1 = x9
           addwf  PRODUCT_L,f  ; Add to lower byte
           btfsc  STATUS,C     ; and any carry to the upper byte
           incf   PRODUCT_H,f
           end
```

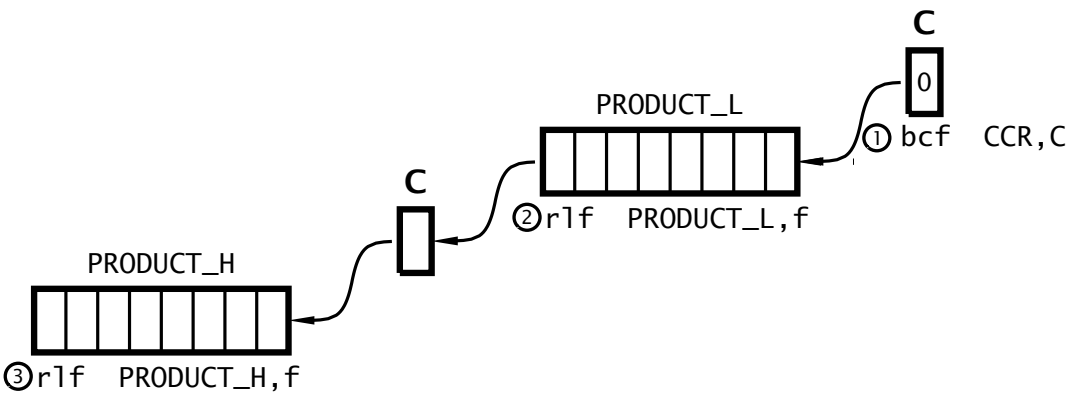


Fig. 3.11 Double-precision shifting.

the `rlf` (Rotate Left File) instruction twice, beginning with the lower byte. As well as shifting the byte `PRODUCT_L` left once, the most significant bit is moved into the Carry flag. Subsequently rotating the contents of the high byte `PRODUCT_H` moves this Carry in as the new least significant bit and shifts the high byte left as shown in Fig. 3.11. Of course the Carry flag needs to be cleared prior to this process. This 3-instruction double-precision routine is repeated three times to give the required $\times 8$ function. Adding the original multiplicand, which is still in `W` gives the final $\times 9$ function.

Example 3.5

The circuit diagram of Fig. 3.12 shows a 7-bit pseudo-random number generator (PRNG) based on a shift register with an Exclusive-OR gate feedback. Devise a routine to continually send these 127 binary random numbers to a port located at `File 06`. The routine must initialize the PRN to any non-zero value.

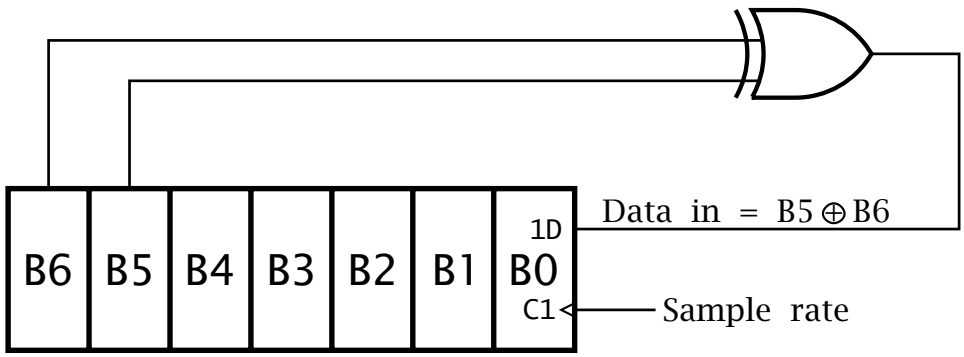


Fig. 3.12 A 7-bit pseudo-random number generator.

Solution

A suitable task list is:

1. Initialize the number to 01.
2. DO forever.
   (a) Rotate shift a *copy* of the number once left so that bits 5 & 6 are aligned.
   (b) Bitwise XOR the number and its shifted copy.
   (c) Rotate the outcome twice left to pop out bit 6 into the **C** flag, which will be B6⊕B5.
   (d) Rotate the original number once left with **C** becoming the new bit 0.

Program 3.8 A 7-bit pseudo-random number generator.

```
PORTB   equ     6
NUMBER  equ     0x20
TEMP    equ     0x21

PRNG    movlw   1         ; Initial value of random number is 01

P_LOOP  movwf   NUMBER    ; Make a copy in memory

        rlf     NUMBER,f  ; Shift number to align bits 5 & 6
        xorwf   NUMBER,f  ; Bitwize XOR them
        rlf     NUMBER,f  ; Shift twice left to put 5XOR6 in Carry
        rlf     NUMBER,f


        movwf   TEMP,f    ; Put original number in memory
        rlf     TEMP,w    ; >> with carry coming in, and put in W

        movwf   PORTB     ; and send this new random number out

        goto    P_LOOP    ; and repeat
```

The listing in Program 3.8 follows the task list fairly closely. The value of the number is temporarily saved in memory so that it can be processed without altering the original number down in **W**. This is done by specifying the destination to be the file, eg. `rlf NUMBER,f`. After rotating left once left, bit 5 in the shifted copy in NUMBER is aligned with the original bit 6 in **W**. After exclusive ORing the two, with the destination again being in File memory (`xorwf NUMBER,f`), bit 6 in NUMBER is now B5⊕B6. Shifting twice left puts this bit in the Carry flag. Finally rotating the original pseudo-random number in **W** moves that pattern once left with the new least-significant bit being the Carry flag, that is B5⊕B6 as specified in the diagram. The first 32 hexadecimal values output are:

02 04 08 10 20 41 83 06 0C 18 30 61 C2 85 0A 14

28 51 A3 47 8F 1E 3C 79 F2 E4 C8 91 22 45 8B 16 ...

The sequence will repeat after 127 output values.

What would happen if the initial value of the random number was zero?

Self-assessment questions

3.1 How could you simply with one instruction toggle bit 0 of any file register?

3.2 As part of a Data memory testing procedure each file in the range File 0Ch through File 2Fh is to be set to the pattern 01010101b (55h). Using Program 3.2 as a model, write a suitable coding to implement this task.

3.3 Write a program to subtract the double-byte datum which is located in File 22:23h, called NUM_2, from NUM_1 in File 20:21h. The double-byte difference is to be in File 24:25h. Remember, if there is a borrow from the lower byte subtraction then an additional one must be subtracted from NUM_1 in the upper byte subtraction. Assume that NUM_2 is smaller or equal to NUM_1. If this were not so how could you determine this situation after the routine has been completed?

3.4 Write a routine that will determine how many hundreds there are in a byte in Data memory at File 20h. The outcome, which is to be in W, will either be 02h, 01h or 00h. For example if the contents of File 20 are FFh (decimal 255) then the outcome will be two. Hint: Try subtracting the number 200 from that in W and examining the Carry/$\overline{\text{borrow}}$ flag. If no borrow then try 100. Again if no borrow then the number must be less than 100.

3.5 The binary approximation to the fraction $\frac{1}{3}$ is:

$$\frac{1}{3} = \frac{1}{2} - \frac{1}{4} + \frac{1}{8} - \frac{1}{16} + \frac{1}{32} - \frac{1}{64} + \frac{1}{128} \cdots$$

Using this series, write a program that will divide a byte in the Working register by three, with the quotient being in the same register at the end. You can use File 20h and File 21h as temporary storage for the quotient and shifting number respectively. The outcome up to $\frac{1}{128}$ is 0.3359375, which is within 0.78% of the exact value. With an 8-bit datum there is no point in including any further elements in the series.

3.6 Write a routine that will count the number of ones in the Working register. For example if W were 01110011b then the outcome in File 20h would be 05h. Hint: Continually shift the number, while incrementing the count when the shifted-out bit in the Carry flag is one. Either do this eight times or else exit when the residue is zero. Remember if taking the latter approach that the Carry flag must be cleared before rotating the number! You may use File 21h as a temporary store for the shifting number.

3.7 Data from an array of data memory between File 30h and File 4Fh is to be transmitted byte by byte to a distant computer over the internet. In order to allow the receiver to examine the data and check for transmission errors it is proposed to append a single byte which is the 2's complement (i.e. the negative value, see page 9) of the 8-bit sum of all the data bytes together. If all the received data bytes plus this **checksum** byte are similarly added then the outcome should be zero if no error has occurred. Code a routine to scan through this data, placing this checksum in File 20h. See Example 3.2 for a template.

3.8 One simple way of encrypting a data byte is to reverse the order of bits. For example 10111100b $\longrightarrow$ 00111101b. Write a routine to implement this reversal on a data byte in File 20h. The encrypted outcome is to be in the Working register. You can use location File 21h as a temporary workspace and W as a loop counter. Hint: Use the Rotate Right and Rotate Left File instruction eight times. If you use W as the loop register then use the instruction `addlw -1` as a decrement W operation.

3.9 Parity is a simple technique to protect digital data from corruption by noise. Odd parity adds a single bit to a word in such a way as to ensure the overall packet has an odd number of 1s. Write a routine that takes an 8-bit byte stored at File 20h and alters its most significant bit to comply with this specification. You can assume that bit 7 is always 0 before the routine begins. Hint: Determine if a binary number is odd or even by counting the number of bits as in SAQ 3.6 and then examining its least significant bit. All powers of two are even except $2^0 = 1$. Thus if this bit is 1 then the number is odd.

PART II

The Software

In Part I we developed the concept of the Havard architecture, ending up with our somewhat simplified BASIC computer. Although BASIC was entirely fictitious, it was designed with an eye to the MCU that forms the basis for the rest of this book.

This part of the text looks mainly at the software aspects of our chosen MCU, the mid-range Microchip PIC family. We will be covering:

- *The internal structure of the MCU.*
- *The instruction set.*
- *Address modes.*
- *The assembly translation process.*
- *Subroutines and modular program design.*
- *Interrupt handling.*
- *The high-level language* **C**.

CHAPTER 4

The PIC16F84 Microcontroller

In this chapter we introduce the PIC16F84 MCU, which we will use as our baseline exemplar for the rest of the text. Here we will primarily look at internal structure, reserving external interfacing considerations for Part 3 of the book.

After reading this chapter you should:

- Recognize the difference between a microprocessor and microcontroller.
- Understand the Harvard-based architecture with its parallel fetch and execute units.
- Appreciate the function, structure and memory map of the unrelated Program and Data stores.
- Be able to interpret the Status register bits that control memory paging and hold the **C**, **DC** and **Z** flags.
- Know how to manipulate the contents of the Program Counter in conjunction with the PCLATH special-purpose file register.
- Understand the interaction between the clock phases and the internal sequence of micro-operations.
- Appreciates the principle of banking in the Data store and its relationship to the RP0 control bit in the Status register.
- Know what peripheral functions are integral to the PIC16F84.

What exactly is a microcontroller unit? In a nutshell, a microcontroller is a MicroProcessor Unit (MPU) which is integrated with memory and input/output peripheral interface functions on the (usually) one integrated circuit. In essence it is a MPU with on-board system support circuitry. Thus we begin by investigating the origins of the MPU. From a historical perspective the story begins in 1968 when Robert Noyce (one of the inventors of the integrated circuit), Gordon Moore[1] and Andrew Grove left the Fairchild Corporation and founded their own company, which they called Intel.[2] Within three years, Intel had developed all the basic types of semiconductor memories used today – dynamic and static RAMs and EPROMs.

[1] Moore's law stated in 1964 that the number of elements on a chip would double every 18 months, although this was subsequently revised to 2 years.

[2] Reputed to stand for INTELligence or INTegrated ELectronics.

As a sideline Intel also designed large-scale integrated circuits to customers' specifications. In 1970 they were approached by the Nippon Calculating Machine Corporation, and asked to manufacture a suitable chip set for a line of calculators to be named Busicom. At that time calculators were a fast-evolving product and any LSI devices were likely to be superseded within a few years. This of course would reduce an LSI product's profitability and increase its cost. Engineer Ted Hoff – reputedly while on a topless beach in Tahiti – came up with a revolutionary way to tackle this project. Why not make a simple computer central computing unit (CPU) on silicon? This could then be programmed to implement the calculator functions, and as time progressed these could be enhanced by developing this software. Besides giving the chip a longer and more profitable life, Intel were in the business of making memories – and computer-like architectures need lots of memory. Truly a brain wave. The Japanese company endorsed the Intel design for its simplicity and flexibility in late 1969, rather than the conventional implementation.

Federico Faggin joined Intel in spring 1970[3] and by the end of the year had produced working samples of the first chip set. This could only be sold to the Nippon Calculating Machine Corporation, but by the middle of 1971, in return for a price reduction, Intel were given the right to sell the chip set to anyone for non-calculator purposes. Intel was dubious about the market for this device, but went ahead and advertised the 4004 "Micro-Programmable Computer on a Chip" in the *Electronic News* of November 1971. The term microprocessor unit was not coined until 1972. The 4004 created a lot of interest as a means of introducing 'intelligence' into electronic products.

The 4004 MPU featured a von Neumann architecture using a four-bit data bus, with direct addressing of 512 bytes of memory. Clocked at 108 kHz, it was implemented with a transistor count of 2300.[4] Within a year the eight-bit 200 kHz 8008 appeared, addressing 16 Kbytes and needing a 3500 transistor implementation. Four bits is satisfactory for the BCD digits used in calculators but eight bits is more appropriate for intelligent data terminals (like cash registers) which need to handle a wide range of alphanumeric characters. The 8008 was replaced by the 8080[5] in 1974, and then the slightly modified 8085 in 1976. The 8085 is still the current Intel eight-bit device. Strangely, 4-bit MPUs were to outsell all other sizes until the early 1990s.

The MPU concept was such a hit that many other electronic manufactures clambered on to the bandwagon. In addition, many designers jumped ship and set up shop on their own, such as Zilog. By 1976 there

[3] He was later to found Zilog (last word (Z) in Integrated LOGic) which became notable with the Z80 MPU – a rather superior Intel 8085.

[4] Compare with the Pentium Pro (also known as the P6 or 80686) at around 5.5 million!

[5] Designed by Masatoshi Shima, who went on to design the 8080-compatible Z80 for Zilog.

were 54 different MPUs either available or announced. For example, one of the most successful families was based on the 6800 introduced by Motorola.[6] The Motorola 6800 had a clean and flexible architecture, could be clocked at 2 MHz and address up to 64 Kbyte of memory. The 6802 (1977) even had 128 bytes of on-board memory and an internal clock oscillator. By 1979 the improved 6809 represented the last in the line of these eight-bit devices, competing mainly with the Intel 8085, Zilog Z80 and MOS Technology's 6502.

The MPU was not really devised to power conventional computers, but a small calculator company called MITS,[7] faced with bankruptcy, took a final desperate gamble in 1975 and decided to make and market a computer. This primitive machine, designed by Ed Roberts, was based on the 8080 MPU and interacted with the operator using front panel toggle switches and lamps – no keyboard and VDU. The Altair[8] was advertised for $500, and within a month MITS had $250,000 in the bank for advance orders.

This first **Personal Computer** (**PC**) spawned a generation of computer hackers. Thus an unknown 19-year-old Harvard computer science student, Bill Gates, and a visiting friend, Paul Allen, in December 1975 noticed a picture of the Altair[9] on the front cover of *Popular Electronics* and decided to write software for this primordial PC. They called Ed Robert with a bluff, telling him that they had just about finished a version of the BASIC programming language that would run on the Altair. Thus was the Microsoft Corporation born.

In a parallel development, 22 Altair owners in San Francisco set up the Home-brew club. Two members were Steve Jobs and Steve Wozniak. As a club demonstration, they built a PC which they called the Apple.[10] By 1978 the Apple II made $700,000; in 1979 sales were $7 million, and then $48 million…

The Apple II was based around the low-cost 6502 MPU which was produced by a company called MOS Technology. It was designed by Chuck Peddle, who was also responsible for the 6800 MPU, and had subsequently left Motorola. The 6502 bore an uncanny resemblance to the Motorola 6800 family and indeed Motorola sued to prevent the related 6501 MPU being sold, as it even had the same pinout as the 6800. The 6502 was one of the main players in PC hardware by the end of the 1970s, being

[6] Motorola was launched in the 1930s to manufacture motor car radios, hence the name "motor" and "ola" – as in pianola. It has the largest share of the world-wide microcontroller market at the time of writing (1999).

[7] Located next door to a massage parlor in New Mexico.

[8] After a planet in *Star Trek*.

[9] The picture was just a mock up, they actually were not yet available; an early example of computer 'vaporware'!

[10] Jobs was a fruitarian and had previously worked in an apple orchard.

the computing engine of the BBC series and Commodore PETs amongst many others.

What really powered up Apple II sales was the VisiCalc spreadsheet package. When the business community discovered that the PC was not just a toy, but could do 'real' tasks, sales took off. The same thing happened to the IBM PC. Reluctantly introduced by IBM in 1981, the PC was powered by an Intel 8088 MPU clocked at 4.77 MHz together with 128 Kbyte of RAM, a twin 360 Kbyte disk drive and a monochrome text-only VDU. The operating system was Microsoft's PC/MS-DOS version 1.0. The spreadsheet package here was Lotus 1-2-3.

By the end of the 1970s the technology of silicon VLSI fabrication had progressed to the stage that several tens of thousands transistors could be integrated on the one chip. Microprocessor designers were quick to exploit this capability in one of two ways. The better known of these was to increase the size of the ALU and buses/memory capacity. Intel were the first with the 29,000-transistor 8086, introduced in 1978 as a 16-bit version of the 8085 MPU.[11] It was designed to be compatible with its eight-bit predecessor in both hardware and software aspects. This was wise commercially, in order to keep the 8085's extensive customer base from looking at competitor products, but technically dubious. It was such previous experience that led IBM to use the 8088 version, which had a reduced eight-bit data bus and 20-bit address bus[12] to save board space.

In 1979 Motorola brought out its 16-bit offering called the 68000 and its eight-bit data bus version, the 68008 MPU. However, internally it was 32-bit, and this has provided compatibility right up to the 68060 introduced in 1995 and ColdFire RISC device launched in 1997. With a much smaller eight-bit customer base to worry about, the 68000 MPU was an entirely new design and technically much in advance of its 80X86 rivals.

The 68000 was adopted by Apple for its Macintosh series of PCs. However, the Apple Mac only accounts for less than 5% of PC sales. Motorola MPUs have been much more successful in the embedded microprocessor market, the area of smart instrumentation from egg timers to aircraft management systems. Of course, this is just the area which MPUs were developed for in the first place, and the number, if not the profile and value, of devices sold for this purpose exceeds those for computers by more than an order of magnitude.

In this applications area an MPU is 'buried' in the application circuit together with memory and various input and output interface circuits. The MPU with its program acts as the controller of the system by virtue of the software in program memory. Over 3.5 billion microprocessor and

[11] and the Intel 8086 architecture-based MPUs are by far the largest selling MPU for computer-based circuitry.

[12] A 2^{20} address space is 1 Mbyte, and this is why for backwards compatibility MS-DOS was limited to 1 Mbyte of conventional memory.

related devices are sold each year for embedded control, making up over 90% of the MPU market.

The second way of using the additional integrated circuit complexity that became available by the end of the 1970s was to keep a relatively simple CPU and use the extra silicon 'real estate' to implement on-board memory and input/output interface. In doing so, simple embedded control systems on the one chip became possible and the overall chip count to implement a given function was thereby considerably reduced. The majority of control tasks require relatively little computing power, but the reduction in size (and therefore cost) is vital. A simple example of this is the intelligent smart card, which has a processor integrated into the card itself. Such microprocessor-based devices were called MicroController Units (MCUs).[13] For example there are about 100 microcontrollers hidden in every home; in the washing machine, microwave oven, telephones, electronic games and so on. About 20 more lurk in the average family car.[14]

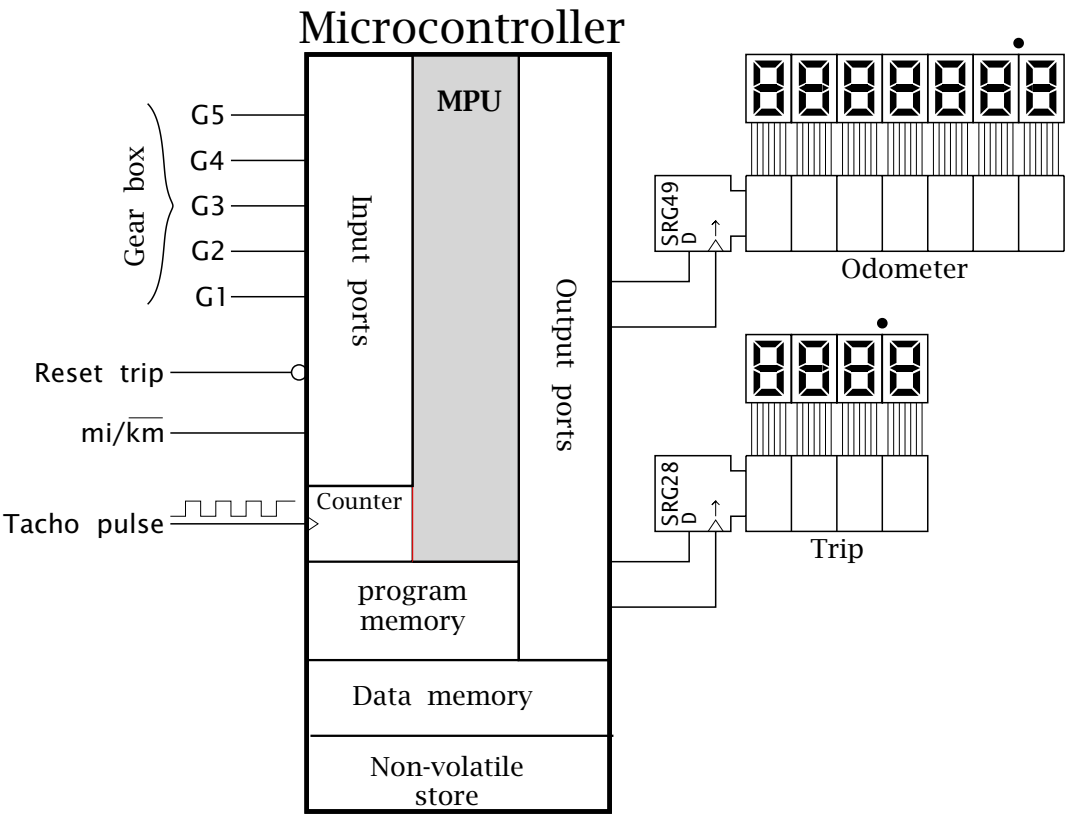


Fig. 4.1 An example of a system based on a microcontroller.

[13] The term *microcomputer* was an alternative term but was easily confused with early personal computers and has dropped into disuse.

[14] New Scientist, vol. 59, no. 2141, 4th July 1998, pp. 139.

In terms of architecture, referring back to Figs. 3.1 and 3.2 on pages 42 and 44 respectively, the microprocessor is the central processor unit, whereas the microcontroller is the complete functioning computer-like system. As an example, consider the electronics of a car odometer monitoring system displaying total distance since manufacture and also a trip odometer. The main system input signal is a tachometer generating pulses on each rotation of the engine flywheel, which when totalized gives the number of engine revolutions – and the pulse to pulse duration could also give the road speed. Of course the actual road distance depends on the gearing of the engine, and thus we need to know which of the five gear ratios has been chosen by the driver at any time. This is shown as five lines G1…G5 originating from the gear box. One signal will be high for the appropriate forward gear, with neutral and reverse being ignored. Additional inputs are used to give a manufacturer's option of a mile or kilometer display, and a user input to reset the trip display to zero.

The display itself consists of seven 7-segment digits (see Fig. 6.6 on page 148) to indicate up to (optimistically) 999999.9. As there are so many segments to control (49 in total), Fig. 4.1 shows the display data fed via a single digital line, shunted serially into a shift register – see Fig. 2.20 on page 36. A second line provides clock pulses for the register with 49 clock pulses being needed to refresh the display.[15]

The trip odometer display comprises four digits, which will record up to 999.9. Similarly two output lines are used to feed and clock the shift register, and 28 clock pulses are needed to shift in a new 4-digit trip display.

The **resource budget** (list of subsystem functions) for this system is:

- An edge-triggered input for the tachometer pulse train, connected to a counter/timer to totalize engine revolutions.
- Seven static digital input lines to interface to the gear ratio, mi/$\overline{\text{km}}$ option and trip reset.
- Four output digital lines to clock the two shift registers and provide segment data.
- A microprocessor to do the calculations and to read/write to the input/output ports respectively.
- Program memory, usually ROM of some kind.
- Data memory for temporary storage of program variables, usually static RAM.
- Non-volatile storage for physical variables, such as total distance and distance since trip reset.

This functionality could be implemented onto a single integrated circuit, and in this situation would be known as a **microcontroller**, that is a mi-

[15] Many displays have this shift register built in as a complete subsystem.

croprocessor integrated with its support circuitry giving a complete microcomputer function. Of course the resource budget listed above is specific to our example. Although the core functions (microprocessor and memory) are common to a wide range of applications, the input/output (I/O) interface needs to be tailored to the task in hand. Some typical I/O functions are:

- I/O to interface to a serial bit stream of various synchronous and asynchronous protocols.
- Counter/timer functions to totalize input events and to generate precision time-varying digital output signals.
- Analog to digital multiplex/conversion to be able to read and digitize analog inputs.
- Digital to analog conversion to output analog signals.
- Display ports to drive multi-digit liquid crystal displays.

This alternative approach to using additional silicon resources led to the first MCUs in the late 1970s. For example the 35,000 transistor Motorola 6801, designed in response to a specific application from an car manufacturer, used the existing 6800 MPU as a core, with 2048 bytes of ROM program memory, 128 bytes of data RAM, 29 I/O lines and a 16-bit timer. With the viability of the MCU approach vindicated, a range of families, each based on a specific core but with individual family members having a different selection of I/O facilities, was introduced by the leading MPU manufacturers. For example, the Motorola 68HC11 family (a development of the 6801 MCU) uses a slightly enhanced 6800 core. The 68HC12 and 68HC16 families use 16-bit cores but are designed to be upwardly compatible with the 8-bit 68HC11. It was quickly realised that many embedded applications did not even need the power of the (antique) 6800 core, and the 68HC05 family[16] had a severely reduced core by lower price. Actually 4-bit MCUs outsold all other kinds of processor until the early 1990s and 8-bit MCUs, now the most popular, are likely to continue in this role for the foreseeable future.

All these MPUs and MCUs were based on the von Neumann architecture (see Fig. 3.1 on page 42) used by mainframe computers. The alternative Harvard architecture (see Fig. 3.2 on page 44), which is chiefly distinguished by having a separate memory space for program and data, originated at Harvard university for a US Defence department computing project, but was rejected in favor of a rival von Neumann design from Princeton university. The first MPU using this architecture was the Signetics 8X300, and this was adapted by General Instruments in the mid 1970s for use as a **Peripheral Interface Controller** (**PIC**) which was designed to be a programmable I/O port for their 16-bit CP1600 MPU.

[16]The 68HC05 has found a niche as the computing engine of smart cards, where high-power computing is not a priority.

General Instruments sold off their microelectronics division in 1988 to a start up company called Arizona Microchip Technology. Microchip's main product was, and is still, a series of microcontroller families based on this PIC architecture. Their first family was introduced in 1989 with the PIC16C5X series. These Harvard processors are based on a set of only 33 instructions. All instructions are coded in a *single* 12-bit word. This use of a primordial instruction set is known as **Reduced Instruction Set Computer** (**RISC**) and contrasts with the **Complex Instruction Set Computer** (**CISC**) model used in most computers/MPUs where several hundred instructions/modes are provided, and because of their number take several memory words to encode. The combination of single-word instructions, the simplified instruction decoder implicit with the RISC paradigm and the Harvard separate Program and Data buses gives a fast, efficient and cost effective processor implementation. The PIC16C5XX 12-bit core family features between 512 and 2048-instruction Program stores implemented as One-Time Programmable (OTP) EEPROM (see page 26), 25 to 73 bytes of Data memory, 12 or 20 I/O pins in the 18- and 28-pin package respectively, and an 8-bit timer. The PIC12CXXX family are 8-pin equivalents.

By 1992 the PIC16CXXX family family based on a 14-bit core enabled easier addressing of larger Program spaces and additional peripheral devices, such as 16-bit timers and A/D converters as well as interrupt handling. The RISC instruction set is virtually identical to the 12-bit core, with a total of 35 instructions. The 16-bit PIC17CXXX core, introduced in 1997, has 58 instruction, with a multiplying ALU and further interface capabilities. It is complemented by the extended 16-bit core PIC18CXXX family introduced in 1999 with 77 instructions more oriented towards high-level language compiler needs.

Of the three families, the 14-bit core is a good compromise between low-cost and ease of use. The PIC16F84, which is the baseline exemplar of this book, is a member of this mid-range family.

From the software point of view all devices with the same core are identical. However, there is a different mix of I/O facilities from the hardware perspective, but with much commonalty. For example, the 16C74 supports an 8-channel analog input port, the PIC16C66 a synchronous serial port and the PIC16F84 a non-volatile data memory. All three devices have similar parallel I/O, timer and interrupt handling facilities.

The architecture of the PIC16F84 is shown in a simplified form in Fig. 4.2. Although initially this looks rather complex, it is little more than the architecture of our BASIC computer of Fig. 3.3 on page 45 but with interface ports connected to the internal File store data bus. You should revise this material now as background to our discussion. In essence the PIC family is based on a Harvard structure with its separate Program and Data store, and with peripheral interface ports mapped onto the Data file

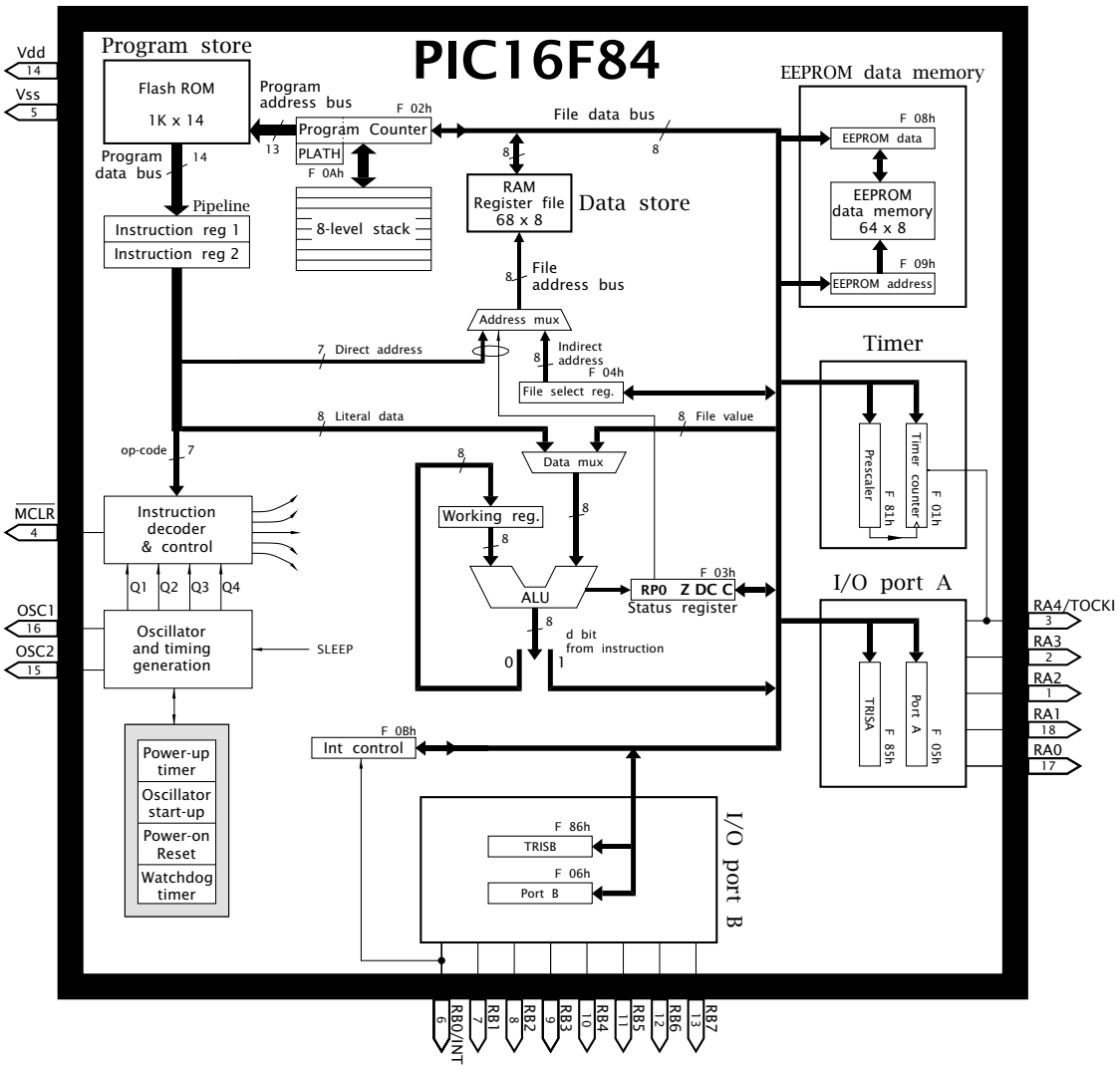


Fig. 4.2 Architecture of the PIC16F84 microcontroller

store address space. That is the various ports appear to the software to be in the Data store. In more detail we have:

Central Processor Unit

As a consequence of the Harvard architecture, the CPU is split between the fetch and execution function, both of which operate in parallel with a minimum of interaction.

Fetch

The fetch section comprises a 13-bit **Program Counter** (**PC**) addressing the Program store via the Program address bus, and a two-deep Instruction pipeline through which 14-bit instructions via the Program data bus progress through to the Instruction decoder – see Fig. 3.3 on page 45.

The Program Counter is actually located in the Data file store at location File 2 and is labelled as **PCL** (**Program Counter Low byte**) in Fig. 4.6.

This means that it can be accessed and manipulated by the software in the same manner as any other file register. For example, if the contents of the Working register were n, then the instruction `addwf 2,f` (see page 49) overwrites File 2 (i.e. the PC) with its original value plus n – that is skip forward n places. A practical example is given in Program 6.4 on page 149.

There is one problem with this example, which arises because the PC is actually 13-bits wide and File 2 only holds the lower eight bits, PC[7…0]. The upper five bits, PC[12…8] are held in a 'buried' register; that is not directly accessible to the programmer. Actually any instruction that directly writes to File 2, such as our example above, does change *all* 13 PC bits as shown in Fig. 4.3. Not only will the 8-bit outcome of the instruction `addwf 2,f` be placed in the lower byte of the PC but the lower five bits of the PC buffer register at File 0A*h*, labelled **PCLATH** (for **Program Counter LATch High byte**) in Fig. 4.3, are automatically copied into the high byte of the Program Counter. The PCLATH file data register is cleared when the PIC is reset and so an instruction like our example will usually result in an address in the first 256-byte 'page' of program memory unless PLATH is loaded with a non-zero value. Thus care needs to be taken when altering the state of the PC by writing to PCL in this manner; especially if the outcome overflows its 8-bit field. Instructions that indirectly alter the value of the PC, such as `goto`, will also use part of the contents of PCLATH in updating the PC. Such instructions carry an 11-bit address as part of the instruction code. Here bits PCLATH[4:3] are moved over to the corresponding PC bits 12 & 11 to give a complete 13-bit PC update – see Fig. 5.4 on page 114.

Associated with the Program Counter is an area of buried storage that can stack up to eight copies of the PC. The current value of the PC is pushed into this **stack** when a subroutine is called or an interrupt is serviced. Conversely a return from a subroutine or interrupt causes the

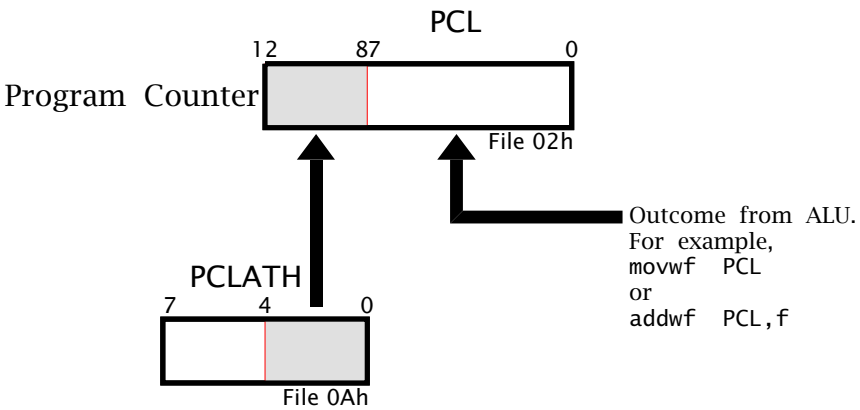


Fig. 4.3 Showing how all 13 bits of the Program Counter are altered when writing to PCL.

last stacked PC value to be popped out again into the PC. Details are given in Chapters 6 and 7.

The PIC microcontrollers have an integral oscillator that generates the internal timing sequences. The oscillator frequency f_{osc} is normally controlled by an external crystal (or ceramic)-capacitor network across the device OSC1 and OSC2 pins – see Fig. 10.4 on page 258. A resistor-capacitor connected to OSC1 may be used as the timing elements where lower precision and frequency stability is not an issue. In this case OSC2/CLKOUT outputs a signal of frequency $\frac{1}{4} \times f_{osc}$. Alternatively an external oscillator can be used as the master clock into OSC1/CLKIN. The PIC16F84 has a maximum frequency f_{OSC} of 10 MHz but there is no minimum. As we shall see (Fig. 10.2 on page 256) the lower the frequency the smaller is the power consumption. Unless otherwise stated, we will assume a f_{osc} of 4 MHz for the rest of the text.

The internal oscillator/clock circuitry must be configured to the appropriate mode, as described in Fig. 10.5 on page 261. This is normally done when code is blasted into the Program store flash EEPROM. When the PIC is powered up (external supply voltage is applied) a 72 ms internal reset pulse is generated by the Power-up timer after the supply rises above approximately 2 volts, followed by 1024 clock pulses, counted by the Oscillator Start-Up timer to ensure that the internal oscillator has stabilised. This latter guard timer only operates for crystal-type oscillator modes – see Table 86 on page 258. The Power-Up timer does not operate if an external reset is applied to the Master CLeaR ($\overline{\text{MCLR}}$) pin.

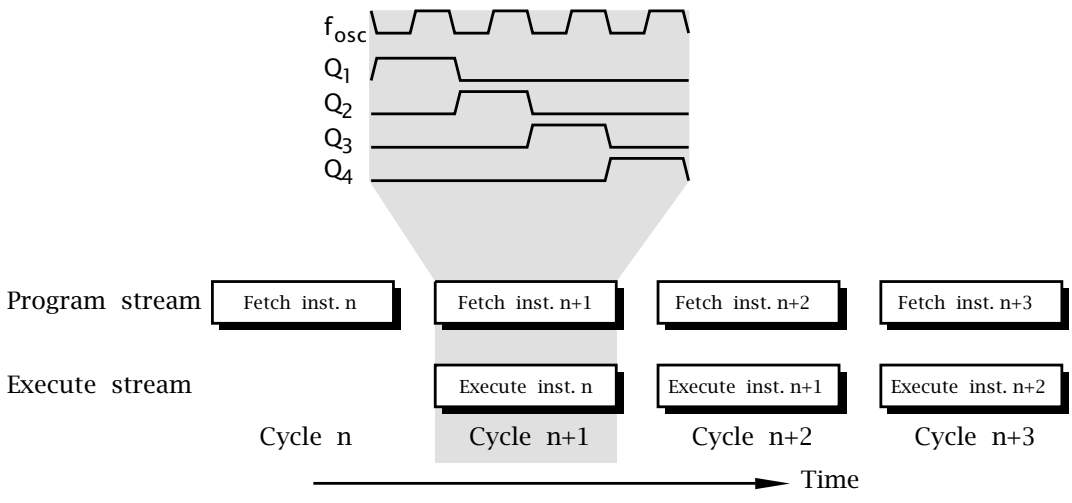


Fig. 4.4 Internal clock sequencing waveforms.

The clock input/crystal frequency at the OSC1 pin is divided by four to generate four internal non-overlapping quadrature clocks, as shown in Fig. 4.4. The clock-related sequence of operations in the fetch unit are:

Q_1: Increment the Program Counter and copy onto the Program store address bus.

Q_4: Read the instruction code off the Program store data bus into Instruction register 1 and at the same time move the previous instruction down the pipeline into Instruction register 2, where it is presented to the Instruction decoder.

Execute

The execution circuitry is centered around the **Arithmetic Logic Unit** (**ALU**) – see Fig. 2.19 on page 35. The ALU processes data from up to two sources. One of these is the 8-bit Working register. The other can be multiplexed either from a file in the Data store or an 8-bit literal, which is part of the instruction code – see page 107. For example `addwf 20h,w` and `addlw 5` respectively add the contents of W to that of File 20h or the constant 5 to W. The outcome can be switched back into W (eg. `addwf 20h,w`) or into the register file (eg. `addwf 20h,f`) as controlled by the single Destination bit in the instruction code – see page 107.

Where an operand is a file, the execution unit can generate the Data store address in one of two ways – see Fig. 4.6.

- Directly via a 7-bit address field in the instruction – see page 107. Seven bits can only directly address up to 128 files. This can be increased to 256 files if the state of the **RP0** (**Register Page bit 0**) bit in the Status register (see Fig. 4.5) is also multiplexed in as part of the Data store address. Where RP0 is zero, then register files 00 – 7F*h* can be addressed, this address page being known as **Bank 0**. With RP0 set to one, then **Bank 1** is addressed, ranging from 80 – FF*h*; see Fig. 4.6. RP0 is cleared when the PIC is reset and the Bit Set File and Bit Clear File instructions may be used alter its state. For example, `bsf 3,5` sets bit 5 (RP0) in the Status register –see Fig. 4.5.
- Indirectly using the File Select Register in conjunction with Indirect addressing, as described in Fig. 3.6 on page 57. In this situation the 8-bit address in the FSR is used to address the Data store whenever the virtual location File 0 is addressed. Here potentially all 256 files in both banks can be addressed irrespective of the state of RP0.

The three flag bits in the **Status register STATUS** are associated with the ALU, giving status information concerning the outcome from an instruction, as shown in Fig. 4.5.

Carry flag

Bit 0 of the Status register is the **C** flag. This primarily holds the Carry out from the last addition operation. Subtraction operations activate this bit as the *complement* of the Borrow out. For example, $24 - 12 = 12\overline{B}1$ and

$12 - 24 = 88\overline{B}0$. **C** also functions as an input/output bit for the Rotate instructions, as shown in Fig. 3.7 on page 61.

The label R/W ? in Fig. 4.5 indicates that this bit can be read from or written to and has an indeterminate value on a power-up reset – its value does not alter on any other type of reset.

Digit Carry flag
Bit 1 of the Status register is the **DC** flag. This operates in the same manner as the standard **C** flag but holds the Carry out from the lower nybble to the upper nybble; that is from bit 3 to bit 4. In the same manner **DC** holds the *complement* of the Borrow out from bit 3 to bit 4. Knowledge of the Carry activity between the lower and upper halves of the byte is useful where binary coded decimal data is being manipulated. Here each nybble holds a 4-bit representation of the decimal digits 0…9 (see page 6) and the half carry then indicates carries between decimal decades.

Zero flag
Bit 2 of the Status register is the **Z** flag. This is set whenever the outcome of the instruction is zero, otherwise it is cleared.

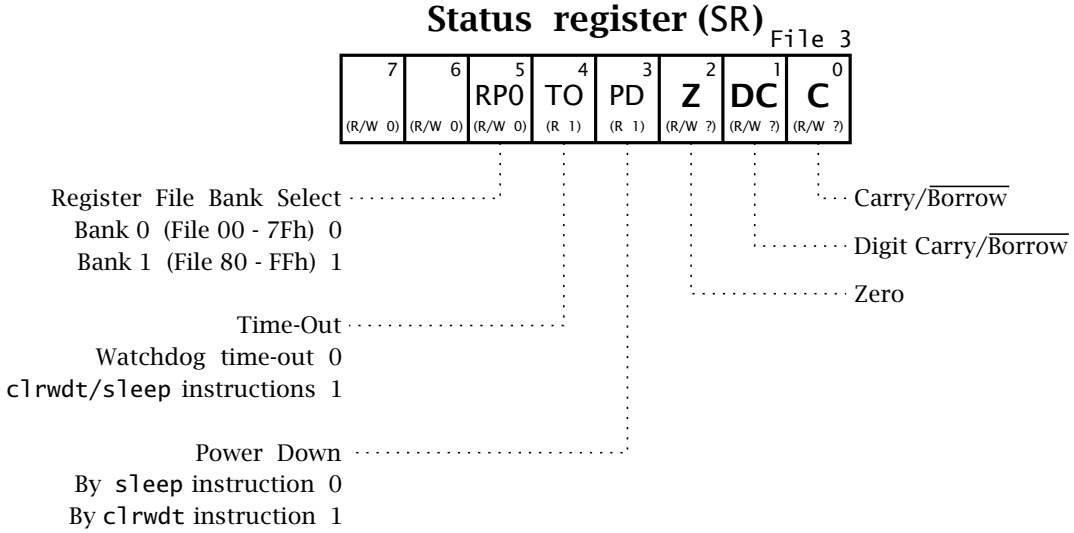


Fig. 4.5 The PIC16F84 Status register

Unlike most MCUs, there are no instructions to specifically clear or set a flag, such as sec for SEt Carry.[17] However, as the Status register is accessible as a file in the Data store, then any instruction that can alter the contents of a file can potentially change the state of a flag. There is a potential problem in that many of these instructions affect one or

[17] For example the Motorola 6800/5/11 families.

more flags (see Table 3.1 on page 53) as part of their execution logic and this *overrides* any change that would result from the outcome of the instruction's execution. For example, `clrf 3` actually sets the **Z** flag to 1. The Bit Clear File and Bit Set File instructions are recommended where an individual bit in the Status register needs to be altered, as these instructions do not inherently affect these flags. For instance, `bsf 3,0` (Set Bit 0 in File 3) is equivalent to `sec` and `bcf 3,2` (Clear Bit 2 in File 3) is equivalent to `clz`.

STATUS also holds the RP0 bank switching bit. The $\overline{\text{TO}}$ and $\overline{\text{PD}}$ read-only bits shown in Fig. 4.5 give information on what type of reset last occurred (Power-Up when power was applied to the device, Watchdog when the Watchdog timer timed out or External by bringing the $\overline{\text{MCLR}}$ pin low) or if awakened from the `sleep` instruction. As these are designated as read-only, they cannot be altered as part of the software, only monitored. These status bits will be discussed in Part 3 of the book.

In the normal Harvard manner, the execution unit is separated from the fetch unit, with distinct data bus, address bus and stores. It is of course controlled via the Instruction decoder which is fed from the bottom of the pipeline in the fetch unit. The fetch Program Counter is in the Data store's address space so the execution unit can effect the fetch sequence by altering the Program Counter, as shown in Program 6.4 on page 149.

The execute unit is also sequenced by the same four clock phases as the fetch unit operating in parallel.

Q_1: Decode instruction.
Q_2: Read from Data store.
Q_3: Process data in ALU.
Q_4: Write into Data store.

Program store
The majority of PIC devices use EPROM for program memory. As EPROM can only be erased using UV radiation (see Fig. 2.11 on page 27) once the software has been programmed into the Program store it can be considered effectively permanently in situ. Such devices are known as One-Time Programmable (OTP). Where it is likely that program code will need to be subsequently altered, the MCU may be housed in a ceramic package with a quartz window, allowing for erasure in around 20 minutes.

An alternative approach is to implement the Program store using EEPROM technology – see page 28. This allows the 'fixed' data to be erased electrically without the expense of a UV transparent package and the time delay inherent with this technology. Using this approach, code can even be reprogrammed in the field, to subsequently upgrade software, without the device having to be removed from the circuit board. Thus, say, a modem's algorithm or a PC's BIOS can be upgraded over a computer network by the user. Microchip's strategy is to substantially increase the

number of EEPROM devices but at the time of writing (2000) most devices are EPROM based.

The 16F84 holds its program code in an internal flash EEPROM (see page 28) memory[18] holding 1024 (2^{10}) instructions, each of 14-bits width. This memory is accessed from the fetch unit via the 14-bit Program data bus into the pipelined Instruction register 1, and is addressed via the Program address bus by the lower ten bits of the Program Counter PC[9…0]. The address range is 000…3FF*h*. As the Program Counter is 13 bits wide, other members of the 14-bit core family can potentially interact with a 2^{13} = 8 kbyte-instruction Program store; for example, the PIC16F876/7 – see Fig. 15.4 on page 440. The structure of 14-bit PIC instructions are discussed in Chapter 5.

All members of the mid-range PIC family use address 000*h* as the Reset vector (the place the PIC goes to when it is reset, for the startup of the program) and 004*h* for the Interrupt vector – the place the PIC goes to whenever it gets an interrupt request.

Data store

The Data store comprises 81 8-bit locations known as **file registers** or just files for short. The contents of any location in the Data store may be moved into or out of the Working register. The PIC16F84's file registers are located in the Data store's memory map as shown in Fig. 4.6. The registers can be categorised as **Special-Purpose Registers** (**SPRs**) used by the core CPU and peripheral modules for status information and controlling the desired operation of the device. The remaining implemented **General-Purpose Registers** (**GPR**) can be used by the programmer for temporary storage of program variables.

There are two ways an instruction can target a datum in the Data store. Each file register has an address, which is listed in Fig. 4.6. For example PORTA is located at File 05.

Directly

Any instruction which can process a datum in the Data store can directly specify the effective address using seven binary bits which are part of the binary program code, the details of which are given on page 107. By itself this can address a base range of 00–7F*h*. However, we see from Fig. 4.6 that this address is augmented by the RP0 bit[19] in the Status register to give an effective 8-bit address. If RP0 is 0 (as it is after reset) the range is 00–7F*h*, that is Bank 0. If RP0 is made 1 then the range is 80–FF*h*; that is

[18] The older PIC16C84 uses normal EEPROM but otherwise has the same Program store architecture. However, the Data store is rather smaller.

[19] Actually the Data store model for the mid-range PICs is based around four 128-byte banks with STATUS[6:5] holding two bank select bits RP1:RP0 (see Fig 5.1 on page 109) to give an effective maximum capacity of 2^9 = 512 register files. Devices such as the PIC16F87X lines implement the complete model. The 8-bit file address for File Indirect memory access is augmented by bit STATUS[7], known as IRP (Indirect Register Page). This gives two 256-byte Indirect pages.

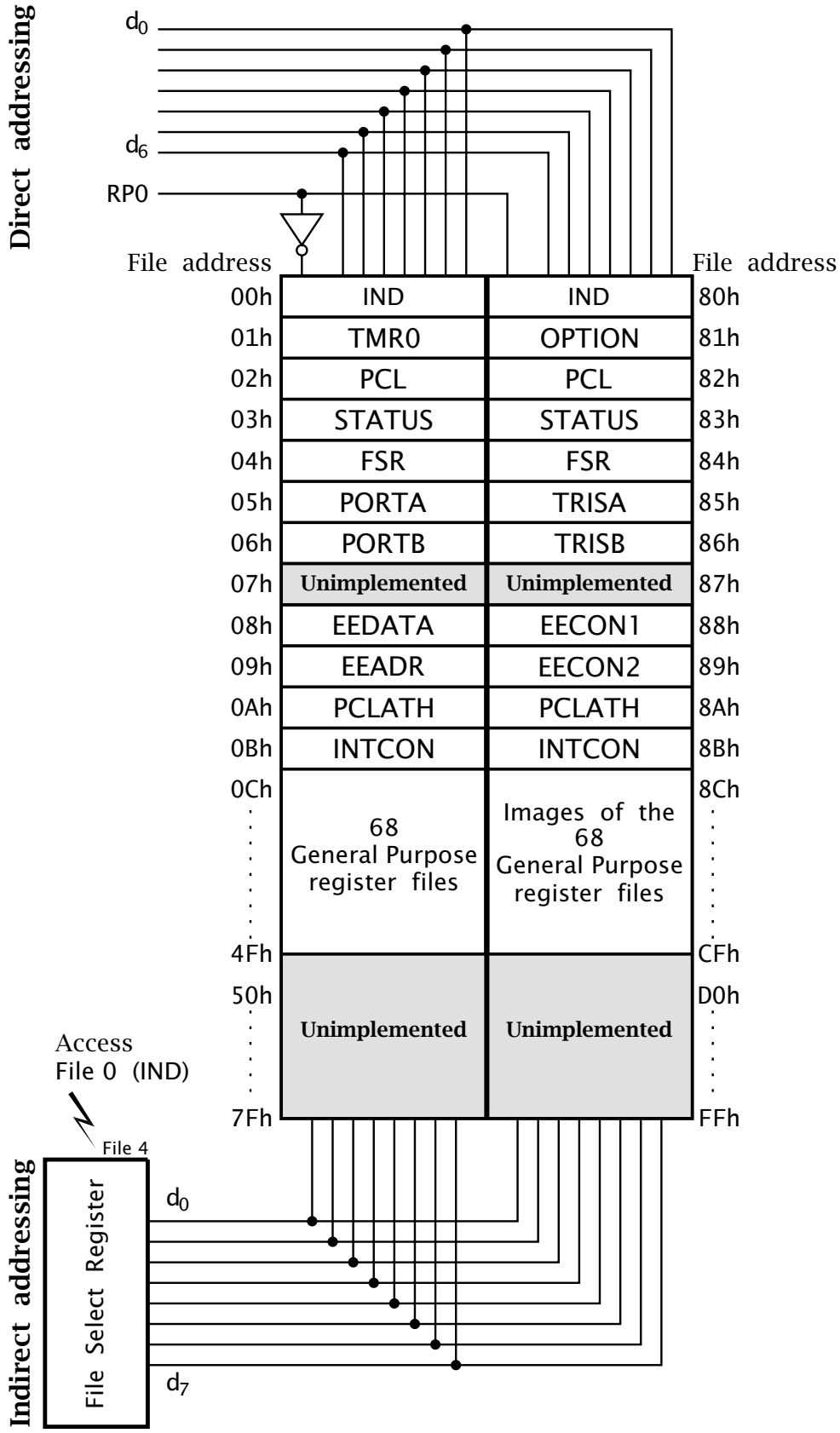


Fig. 4.6 Data store memory map.

Bank 1. For example, to set the contents of File register TRISB (File 86*h*) to all 1s and then to read the contents of the File register PORTB into the Working register, we need the following program:

```
PORTB  equ   06        ; Register file in Bank 0
TRISB  equ   86h       ; Register file in Bank 1
STATUS equ   03        ; Status register
RP0    equ   5         ; RP0 is bit 5 in STATUS
; ----------------------------------------------
       bsf   STATUS,RP0; Set RP0 to 1 -- select Bank 1
       movlw 0FFh      ; Bit pattern 11111111b
       movwf TRISB     ; Sent to TRISB
       bcf   STATUS,RP0; Select Bank 0
       movf  PORTB,w   ; Read the state of PORTB
```

Indirectly

If the Indirect address mode is used, as was described in Fig. 3.6 on pageref3:fig-Indirect, then the 8-bit address in the File Select Register (FSR) is used as the effective. As we have an 8-bit address in this situation any location in the two banks are accessible with not bank switching required. The same example as described above is then implemented as:

```
PORTB  equ   06        ; Register file in Bank 0
TRISB  equ   86h       ; Register file in Bank 1
STATUS equ   03        ; Status register
RP0    equ   5         ; RP0 is bit 5 in STATUS
; ----------------------------------------------
       movlw TRISB     ; Set up the pointer to locate TRISB
       movwf FSR
       movlw 0FFh      ; Bit pattern 11111111b
       movwf INDF      ; Sent to TRISB
       bcf   STATUS,RP0; Select Bank 0
       movf  PORTB,w   ; Read the state of PORTB
       movlw PORTB     ; Set up the pointer to locate Port B
       movwf FSR
       movf  PORTB,w   ; Read the state of PORTB
```

Although there is no Bank switching required, this code segment is actually longer than the previous solution! However, Indirect addressing is useful when one location in Bank 1 requires frequent access.

The bottom 12 locations of both banks are reserved for SPRs. Although the exact location can vary across members of the mid-range family, common registers; for example, PCL and PORTA tend to have the same location. For instance, see Appendix B.

Of these, we have already met most of those involved with the core function:

INDF

The INDirect File at File 0 is not physically implemented as a register. Instructions accessing this virtual location actually put the contents of the FSR onto the Data store address bus, as described in Fig. 3.6 on page 57.

PCL
The Program Counter Low byte is addressed as File 2. Its relationship with the total 13-bit PC is described on Fig. 4.3.

STATUS
The Status register can be accessed in File 3. As can be seen from Fig. 4.5, this file holds the three code condition bits plus several status bits and the Data store page bit RP0.

FSR
The File Select Register at File 4 holds the indirect address used when the instruction refers to the virtual INDF address.

PCLATH
File 0A*h* holds the LATch High byte for the Program Counter, as described in Fig. 4.3.

INTCON
The INTerrupt CONtrol register at File 0B*h* holds the mask and status bits controlling the response of the MCU to interrupts. Its operation is described in Chapter 7. Most devices have interrupt-related bits in other registers – for example see Fig. 14.10 on page 408.

All these core SPRs are images in both memory banks.

The remaining nine SPRs relate to the configuration and control of the various peripheral interface devices. More details will be given on individual peripheral SPRs in Part 3 of the book in the appropriate chapters.

The 68 GPRs are located from File 0C*h* through File 4F*h* and are mirrored in Bank 1.[20] Thus the instruction `clrf 4Fh` and `clrf 0Cfh` are identical and target the same physical location irrespective of the state of the Register Page bit RP0 setting. The remaining file locations are not implemented and read as 00*h*, as does location File 07h/File 87*h* – which is reserved for PORTC and TRISC in devices with 28+ pins.

Peripheral functions
Each member of the PIC family has its unique set of integrated peripheral devices. However, all PICs have parallel input/output and timer facilities. As well as these standard facilities, the PIC16F84 has a peripheral 64-byte EEPROM which can be used as a small data store not dependent on continuous power to retain its contents; i.e. non-volatile.

Each of these peripheral facilities are described in detail in Part 3 of the book, but for completeness are briefly cataloged here together with their associated SPRs. These registers are used to configure the function of their target peripheral interface, to control and monitor their status.

Parallel input/output
The ability to externally alter or monitor several digital lines at the same time is a virtually universal facility on microprocessor-based systems.

[20] This GPR mirroring is not as a rule a feature of the more sophisticated members of the family.

Apart from the 8-pin PIC12XXX series, all PICs have a minimum of 12 such external input/output lines. Some have much more, such as the 40-pin PIC16C74 which has 33 I/O lines.

The PIC16F84 has 13 I/O lines, divided up into two ports. Port A has five I/O lines mapped into the Data store address space at File 5. The remaining eight lines are allocated to Port B at File 6. These ports can be thought of as a 'window' into the Data store in that data written to File 5 or File 6 appear to the outside world on the corresponding pins; pins RA4...RA0 and RB7...RB0 respectively – see Fig. 10.1 on page 254. However, the electrical and logic behavior of these ports is more complex than that of a purely internal register file. This will be discussed in Chapter 11 but as an example a port bit must be configurable as either an output (so that the CPU can control the state of the associated pin) or an input (so that the CPU can read the state of this pin). To do this, each Port register has an associated Data Direction register, which Microchip call TRISA and TRISB, which map to File 85h and File 86h respectively (the term TRIS stands for TRIState – see Fig. 11.2 on page 273. These registers lie in the less convenient Bank 1 as they are usually set up at the beginning of the program and never subsequently altered.

As an example, consider that we wish to make Port B bits 6...0 an input and bit 7 an output. Then the setting up code would be:

```
STATUS equ   03          ; Status register is at File 03
RP0    equ   05          ; Bank switch bit is 5 in STATUS
TRISB  equ   86h         ; Data Direction register at File 86h
PORTB  equ   06          ; Port B itself is at File 06

       bsf   STATUS,RP0  ; Bank 1
       movlw 7Fh         ; Binary pattern 0111111
       movwf TRISB       ; makes RB7 output, RB6...RB0 input
```

As an example, subsequently pin RB7 can be pulsed high and then low as follows:

```
      bsf   PORTB,7      ; Pin RB7 high (set bit 7)
      nop                ; Delay a short time
      nop                ; by putting in a few NO Operations
      bcf   PORTB,70      ; then low (clear bit 7
```

with the assumption that the CPU is still in Bank 0.

The registers associated with parallel I/O are:

PORTA, File 05h

Only the lower five bits are implemented in this register file, feeding through to pins RA4...RA0. Pin RA4 is shared with the Timer peripheral. The upper three bits read as zero.

TRISA, File 85h
This is used to bitwise configure Port A bits as input or output. Setting TRISA[n] to 1 sets bit PORTA[n] as an input and to 0 as an output. Any type of reset sets the TRIS bits to 1 and the associated port bits to input.

PORTB, File 06h
A bi-directional 8-bit port connected to pins RB7...RB0. Bit RB0 doubles as a hardware interrupt input.

TRISB, File 86h
This is used to bitwise configure Port B bits as input or output. Details are the same as TRISA.

Timer
Most MCUs have facilities to either measure elapsed time and/or to generate digital on/off waveforms with well defined durations. This is normally based around one or more counters that are incremented either from an external pulse or internal clock. For instance, if an automatic packing machine needs to count cans of beans going along a conveyer belt, then a photoelectric-based transducer could act as the timer input. If a new packing carton needed to be in place every 24 cans then an internal 8-bit counter would be set to E8h (-24). When the counter overflows from FFh to 00h then an interrupt (see Chapter 7) would be generated and the MCU then take the appropriate action.

All PIC MCUs have at least a basic timer/counter known as Timer 0 (TMR0). The read/write TMR0 counter register at File 1 can be clocked from the outside world via the T0CKI (Timer 0 ClocK In) pin, which is shared with the RA4 Port A pin. Alternatively, the incrementing source can be the internal Q_4 phase clock, which is one quarter of the crystal frequency. For example, for a 4 MHz crystal this is 1 MHz. Either clock source can be frequency divided by a buried 8-bit prescale counter. This divide ratio is controlled by the lower three PreScale bits of the OPTION_REG register at File 81h (see Fig. 13.2 on page 363), labelled PS2:PS1:PS0. The ratio is then 2^{PS+1}. For example, if PS[2 : 0] = 111 then the counter will increment at $\frac{f}{256}$, where f is the clock source frequency. The prescaler can be disconnected by setting bit 3 of OPTION_REG to 1. This will give a direct connection between pulse source and counter. Writing to the Timer 0 register also zeros the Prescaler counter (for example `movlw 0F8h, movwf 1`) enabling the time period to begin from true zero time.

When this PSA (Pre-Scale Assignment) bit is 1 the prescaler becomes a postscaler to the **Watchdog timer** – see Fig. 13.1 on page 362. The Watchdog timer is designed to reset the MCU unless periodically preset by the user's program with the instruction `clrwdt` (CLeaR WatchDog Timer). This ensures that the PIC will eventually reset if due to an electrical disturbance or a software bug, the processor malfunctions, perhaps

by jumping into an unprogrammed part of the Program store. This will disrupt the periodic preset. If the prescaler is assigned to the Timer then the Watchdog timer will periodically time-out (count down through zero) after approximately 18 ms.[21] With PSA set to 1 then 2^{PS} 18 ms Watchdog time-outs are required before the processor is reset. Thus, with PS[2 : 0] = 111, $2^7 = 128$ time-outs gives a period to MCU reset of nominally 2.3 s. Thus the software must use the `clrwdt` instruction before this period elapses to prevent reset. This instruction also clears the Prescale counter. If it does time-out, then the TO bit in the Status register will be cleared. If desired the Watchdog timer may be disabled at the same time as code is programmed into the Program store. Various configuration bits (known as fuses) are located by the flash EEPROM programmer in location File $2007h$ (see Fig. 10.5 on page 261), which is not accessible during the normal run mode. Such details are normally hidden to the operator by the EEPROM programmer's operative software.

Registers relating to Timer 0 are:

TMR0, File 1
Sometimes known as the Real-Time Clock/Counter (RTCC), is an 8-bit up-counter register that keeps tally of clock events. It may be preset to any byte value by moving data from W and read at any time. When it overflows from FFh to 00h it sets the T0IF (Timer 0 Interrupt Flag) in the INTCON (INterrupT CONtrol) register – see Fig. 7.4 on page 178. This may be used to generate an interrupt.

OPTION_REG, File 81h
Six bits in this register in Bank 1 at File 81h are used in conjunction with the timer – see Fig. 13.2 at page 363.

- PS2, PS1, PS0 at bits 2,1,0 respectively control the prescale ratio 2^{PS-1} for the timer or postscale ratio 2^{PS} for the Watchdog timer.
- T0SE (Timer 0 Set Edge) at bit 4 allows the programmer to select which edge of a pulse at the T0CKI pin will increment the counter; a 0 for _/‾ and 1 for ‾_.
- T0CS (Timer 0 Clock Select) at bit 5 allows the programmer to select the clock source as either the internal clock (= 0) or a transition at the T0CKI pin.

The remaining two bits configure external interrupt edge select and electrical properties of Port B inputs.

Data EEPROM
The PIC16F84 has a block of 64 bytes of data that does not require power to retain its contents. This **non-volatile memory** is not part of the (volatile) Data store and is accessed through SPRs as a peripheral device. Any byte can be addressed and then read from or written to via the

[21] Time-out is *very* nominal due to process variations, 7 – 33 ms, and in addition is rather dependent on supply voltage and temperature.

EEDATA register as addressed by the EEADR register and controlled by the EECON1 and EECON2 control file registers. Data EEPROM has a minimum endurance of 1,000,000 writes and such data is retained for upwards of 40 years. Some typical uses of a non-volatile depository would be to hold the number of pages printed in a laser printer or total miles/kilometers travelled in a car.

Details of the Read and Write protocols are given in Chapter 15, but are briefly reviewed here for completeness.

Read

1. Put address (00 – 3F*h*) into EEADR.
2. Set RD (bit 0 of EECON1) to 1 to set to the ReaD mode.
3. Read the addressed contents in EEDATA.

Write

1. Put address into EEADR.
2. Put data into EEDATA.
3. Set WREN (bit 2 of EECON1) to 1 to WRite ENable.
4. Put code 55*h* into EECON2.
5. Put code AA*h* into EECON2.
6. Begin the Write cycle by setting WR (bit 1 of EECON1) to 1.

Writing, which is normally an infrequent act, is deliberately made circuitous to protect against accidental changes to the EEPROM. The register EECON2 does not actually exist, but the interlock writing 55*h* followed *directly* by AA*h* to File 89*h* is a necessary part of unlocking the target byte. Interrupts can disrupt this sequence and should be inhibited if used. Writing takes around 50 ms to complete, and sets the EEIF (EEPROM Interrupt Flag) bit 4 of EECON1 after this time, and this can be used to interrupt the processor. The WRERR (WRite ERRor) bit 3 of EECON1 is set if a Write cycle is prematurely terminated, say, by an External reset.

Registers associated with the Data EEPROM are:

EEDATA, File 08*h*

This contains the addressed data after a Read action and holds data to be written into the addressed byte during a Write action.

EEADR, File 09*h*

The 6-bit address of the target byte is placed here before a Read or Write cycle.

EECON1, File 88*h*

This holds the control and status bits that:

- Trigger an EEPROM Read.
- Enable a Write action.

- Trigger an EEPROM Write.
- Signals a premature end to a Write cycle.
- Signals a Write cycle has been completed.

Details are given in Fig. 15.2 on page 434.

EECON2, File 89h
The EEPROM CONtrol 2 is not a physical register and reads as zero. This address is used as the target for the Write cycle unlocking sequence which is implemented by moving 55h followed directly by AAh into this virtual location.

Examples

Example 4.1
Discuss how the performance of the PIC architecture is improved by incorporating pipelining into the design of the instruction-fetch unit. Do you forsee any problems associated with handling Jump instructions (such as `goto`) in connection with the pipeline structure?

Solution
The pipeline is a precondition for the parallel operation of the fetch and execution units. That is, in order to allow the execution of instruction n whilst the next instruction $n + 1$ is being fetched from the Instruction store, internal storage must be provided to present the instruction code to the Instruction decoder. As all instructions are the same size, that is 14 bits, then the pipeline register structure and control is considerably simplified. Most conventional CISC processors have instructions that vary considerably in length. For example the 68HC11 MCU core has instructions that cover the range one through four bytes; that is the fetch phase can take between one and four bus transactions. Some more sophisticated processors have multi-stage pipelines with each stage feeding part of the execution circuitry. Thus several streams of execution activity can occur simultaneously.

The problem with pipelines is that they presuppose that the program instructions will be executed sequentially as they are stored in memory. However, instructions that disrupt this smooth running and move on the Program Counter require that the pipeline be emptied so that the instruction code of the destination travels down to the end of the pipe. For example, if instruction k is `goto n`, then instruction $k + 1$ will be in the first stage of the pipeline by the time the processor knows that the next step is actually to be instruction n. Thus a null instruction cycle needs to be executed which simply brings this instruction code into the pipeline but does not execute instruction $k+1$ whose code is at the end of

the pipeline. This is sometimes known as **flushing** the pipeline. Instructions such as `goto` need two clock cycles to execute. Conditional Skip instructions, such as `incfsz` and `btfsc` take two cycles when the skip is implemented and one otherwise. All other instructions always take one cycle.

Example 4.2

Can you determine why after a subtraction, or addition of a negative number (eg. `addlw -6`), the setting of the **C** flag is the *complement* of the borrow-out. Hint: Look at 2's complement arithmetic on page 9.

Solution

The solution lies in the method of subtraction by 2's complementing the subtrahend. The subtract instructions convert the subtrahend to their 2's complement form and then configure the ALU to add. After the addition there can be two outcomes, depending on the relative magnitude of the minuend and subtrahend.

1. Where the subtrahend is greater than the minuend then the outcome is negative and there is no carry-out. An example of this situation is: $06 - 0A \rightsquigarrow 00000110 + 11110110 = (0)\ 11111100$ or -4 (no carry).
2. Where the subtrahend is less than the minuend then the outcome is positive and there is a carry-out. An example of this situation is: $0A - 06 \rightsquigarrow 00001010 + 11111010 = (1)\ 00000100$ or $+4$ (carry).

In both cases the Carry flag acts as an inverted borrow. This is in keeping with the RISC philosophy of the PIC family, to keep the processor 'lean and mean'. In any case this non inversion means that subtraction can be implemented by adding negative data, eg. `addlw -6`. This is translated by the assembler to `addlw 0FCh`, where FCh is of course the 2's complement of 6.

Example 4.3

Write a program to increment a packed BCD quantity located in Data memory at File 20h.

Solution

Two Binary-Coded Decimal (BCD) digits may be packed into a single byte to represent numbers up to 99. For example $\boxed{0100\ 1001}_{\text{File 20h}}$ represents BCD 49. Incrementing a number stored in this hybrid decimal-binary form using the normal binary addition rules may give an incorrect result. For example 01001001 + 1 (49 + 1) gives 01001010 (4Ah) after addition, but should give 01010000 (50h). Similarly, 10011001 + 1 (99 + 1) gives 10011010 (9Ah) instead of 00000000 Carry 1 (1 00h).

From these examples it can be seen that whenever any of the BCD decades equals ten after incrementation then it should be zeroed and one added to any higher decade. Based on this increment and add algorithm we can formulate the task list.

1. Increment the packed BCD byte using normal binary arithmetic.
2. IF the lower nybble of the outcome is ten then add six to the outcome.
3. IF the upper nybble of the outcome is ten then add six to it.

Program 4.1 Incrementing a packed BCD byte.

```
;*****************************************************************
;* FUNCTION: Increments a BCD datum giving a BCD outcome         *
;* ENTRY   : BCD in F20h                                         *
;* EXIT    : BCD+1 in F20h                                       *
;* EXAMPLE : 10011000 (98) + 1 = 10011001 (99)                   *
;* ***************************************************************
;
STATUS     equ   3          ; The Status register
C          equ   0          ; Carry flag is bit 0
DC         equ   1          ; Digit Carry flag is bit 1
BCD        equ   20h        ; The BCD number is in File 20h
; -------------------------------------------------------
BCD_INC    incf  BCD,w      ; Binary inc BCD number and put in W
           addlw 6          ; Add six
           btfss STATUS,DC  ; Needed IF produced a half carry
           addlw -6         ; ELSE not needed
; Now check the upper digit by adding 6 to it and checking Carry
           addlw 60h        ; Add 60h (ie six to upper digit)
           btfss STATUS,C   ; Needed IF caused a Carry
           addlw -60h       ; ELSE cancel the correction factor
; The incremented and corrected BCD number is now in W
           movwf BCD        ; Put it out in memory
END        ..... .....
```

Program 4.1 gives an efficient implementation of this task list. After incrementing using normal binary rules, six is added to the previous outcome and the **DC** flag checked for activity. This flag will only be set when the original nybble was ten ($0A_h + 6 = 10_h$). In this case the add six operation is allowed to stand as the necessary correction otherwise it is cancelled by subtraction. The upper nybble (BCD digit) is checked and corrected in the same manner, but this time it is the full Carry flag that is tested. If this is set, then the addition of 60_h is allowed to stand, otherwise it is subtracted. This Carry flag could be used to set a hundreds digit if desired, to show overflow from 99 to 100.

An alternative approach would be to subtract nine *before* incrementation and if the **Z** flag is set then leave the digit at zero and increment the higher digit; otherwise add ten. Repeat for the upper digit.

Example 4.4

Write a routine that will add two packed BCD numbers at File $20h$ and File $21h$. The outcome is to be in File $22h$ for the hundred's digit and File $23h$ for the tens and unit digits.

Solution

As in the case of Example 4.3 the binary addition of numbers that are already in BCD form may need correction afterwards. Again, a correction will be needed where the digit outcome is greater than nine. The situation is more complex in this case as the sum of two digits can be anywhere between zero and 19 ($9+9+1$(carry-in) $= 19$). The illegal range 10 to 19 may be determined by applying the criterion:

- Check for the range Ah to Fh (10 to 15) — for example, $3 + 9 =$ Ch ($3 + 9 = 12$).
- Otherwise check if there was a carry out to the next decade — for example, $9 + 9 = 1\ 2h$ ($9 + 9 = 18$).

In both cases the digit may be corrected by adding six to 'jump over' the six illegal BCD states.

Based on this course a possible task list is:

1. Add the two packed BCD bytes using normal binary arithmetic.
2. IF the addition results in a full Carry THEN Hundreds = 1
3. IF the addition results in a half Digit Carry OR IF the unit BCD nybble is greater than nine THEN add six to the outcome.
4. IF the last correction (if any) resulted in a full Carry OR IF the tens BCD nybble is greater than nine THEN add six to the upper nybble of the outcome.

Program 4.2 directly implements this strategy. The full Carry flag is tested three times.

1. Directly after the addition – for example, $99h + 99h = 1\ 32h$.
2. After the first correction where six is added to the lower digit – for example, $77h + 88h =$ FFh; FF$h + 06h = 1\ 05h$.
3. After the second correction where six is added to the upper digit – for example, $70h + 80h =$ F0h; F0$h + 60h = 1\ 40h$.

If the flag is set at any point then the hundreds digit will be one; the maximum value in this case is $99 + 99 = 198$.

A nybble is tested for over nine by subtracting ten. If a carry (half or full depending on which digit is being tested) is not generated then the original nybble must have been ten or greater. I have actually implemented the subtract ten operation by adding -10; that is `addlw -10`. This is because the `sublw` instruction actually takes the value in the Working register away from the literal and not the other way around – see also page 55.

Program 4.2 Adding two packed BCD numbers.

```
;*****************************************************************
;* FUNCTION: Adds two packed BCD datums giving a BCD outcome      *
;* ENTRY   : BCD_1 in F20h, BCD_2 in F21h                         *
;* EXIT    : SUM_H (hundreds digit) in F22h                       *
;* EXIT    : SUM_L (tens:units digit) in F23h                     *
;* EXAMPLE : 10011001 (99) + 10011001 (99) = 00000001 10011000 (198) *
;* ***************************************************************
;
BCD_1      equ    20h        ; The first tens:units BCD number
BCD_2      equ    21h        ; The second tens:units BCD number
SUM_H      equ    22h        ; The hundreds digit of the outcome
SUM_L      equ    23h        ; The tens:units digits of the outcome
STATUS     equ    03h        ; The Status register
C          equ    0          ; holding the Carry flag at bit0
DC         equ    1          ; and the Digit half Carry at bit1
; Task 1: Add the two numbers using natural binary rules

BCD_ADD    clrf   SUM_H      ; Zero the hundreds digit
           movf   BCD_1,w    ; Get the first packed BCD number
           addwf  BCD_2,w    ; Add the second number
           movwf  SUM_L      ; and put away as the uncorrected result

; Task 2: Was there a full Carry from this binary addition?
           rlf    SUM_H,f    ; Rotate Carry into hundreds digit

; Task 3: Was there a Digit half Carry from the binary addition?
           btfsc  STATUS,DC  ; IF DC=0 THEN skip
            goto  ADJUST_1   ; ELSE an adjustment is needed

; or if not, is the unit BCD digit greater than nine?
           addlw  -0Ah       ; Take away ten
           btfss  STATUS,DC  ; Did this give a Digit carry?
            goto  TENS_CHECK ; IF it did THEN not >9
; -----------------------------------------------------------------
ADJUST_1   movlw  6          ; Correct the units digit by adding six
           addwf  SUM_L,f    ; with the adjusted value back in the file
           btfsc  STATUS,C   ; IF this gives a full Carry
           incf   SUM_H,f    ; THEN record a hundreds digit
; -----------------------------------------------------------------
; Task 4: Was there a full Carry from the previous correction (if any)?
TENS_CHECK btfsc  SUM_H,0    ; IF yes THEN hundreds digit will be 01h
            goto  ADJUST_2   ; and the tens digit will need adjustment

; or IF not, is the tens BCD digit >9?
           movf   SUM_L,w    ; Get the current partly corrected sum
           addlw  -0A0h      ; Take away ten from the tens digit
           btfss  STATUS,C   ; Did this give a Carry?
            goto  FINISH     ; IF it did THEN not >9
; -----------------------------------------------------------------
ADJUST_2  movlw   60h        ; Correct by adding 6 to the tens digit
          addwf   SUM_L,f    ; with the adjusted value back in the file
          btfsc   STATUS,C   ; IF this generates a Carry
          incf    SUM_H,f    ; THEN record a hundreds digit
; -----------------------------------------------------------------
FINISH    ....    .....
```

Self-assessment questions

4.1 The PIC16F877 mid-range MCU has a 8 kbyte Program store, which can hold up to 8192 14-bit instructions located in the range 0000 – 1FFFh. As part of a certain program, execution has to be transferred from the bottom quarter of this map to 1000h. Given that the goto instruction can only directly access 11-bit addresses (0000 – 07FFh), how could you engineer a jump to this address, i.e. goto 1000h?

4.2 Write a routine to decrement a packed BCD quantity, as in Example 4.3. Hint: Devise a simple test to activate the relevant carry flag if the digit is Fh.

4.3 Write a routine to add ten onto a packed BCD byte located in File 20h. Hint: This is similar to Example 4.3 but only the tens digit is augmented.

4.4 Where microprocessors are used in a general-purpose computing environment, the program is normally loaded into and run from read/write RAM memory. This means that the system can run a word-processor one minute and a spreadsheet program the next. Of course this means of operation is not applicable to embedded applications, where the program is stored in some variety of non-volatile read-only memory. Discuss why this is so and the virtues of ROM, EPROM and EEPROM implementations of non-volatile storage.

4.5 The following routine is designed to do nothing other than defer execution of the main routine by 1282 μs in a PIC MCU which is being clocked using a 4 MHz crystal. Taking into account pipelining, verify this figure.

```
DELAY   clrf    20h     ; Clear count
D_LOOP  incf    20h     ; Increment
        decf    20h     ; and cancel by decrementation
        decfsz  20h     ; Decrement until zero
         goto   D_LOOP
        ....    .....
```

4.6 Can you devise an algorithm and task list to subtract two packed BCD numbers located as in Example 4.4. The outcome packed byte is to be in File 23h and File 22h is to hold the overflow carry – that is the negative indicator. This overflow is to be zero if there was a full borrow out, otherwise one. For example 45h − 29h = (01)16h (that is 45 − 29 = +16) and 29h − 45h = (00)84h (29 − 45 = −84) where 84 is the ten's complement of 16.

CHAPTER 5

The Instruction Set

If you like to think of writing a program as analogous to preparing an elaborate meal, then for any given cooking appliance, such as a microwave oven or electric stove (the hardware) there are a range of processes. These processes – for example, steaming, frying, boiling – are analogous to the **instruction set** which can be implemented by the CPU. The various ingredients that can be handled by a process are the instruction's data. Such data may lie in an internal register or out in the Data store. There are several different ways of specifying the **effective address** (**ea**) of an operand. These are known as **address modes**.

In keeping with the PIC microcontrollers' RISC-like philosophy, the mid-range core have a total of only 35 instructions. Each instruction code is contained in a 14-bit word which holds the instruction operation code, address or data and destination bit. We have covered most of these instructions and address modes when discussing our BASIC computer back in Chapter 3; now would be a good time to go over this material. As we will begin this chapter by examining PIC's address modes and how they are incorporated into an instruction's binary word, we will use BASIC's instruction set listed in Table 3.1 on page 53 for our illustrative examples. The latter part of the chapter looks at the full instruction set in some detail.

After reading this chapter you will:

- Know that an address mode is the way an instruction pin-points its data.
- Understand how Inherent, Literal, Register Direct, File Direct, File Indirect, Bit and Absolute address modes permit an instruction to target an operand for processing.
- Know that Movement instructions, copying data in-between the Working register and the Data store, are the most used of the instruction categories.
- Appreciate that the processor can directly implement the common arithmetic operations of Addition, Subtraction, Incrementation and Decrementation.
- Know that data in the Data store can be rotate-shifted through the **C** flag.

- Understand how to use the four basic logic instructions to invert, set, clear, toggle, bit test and differentiate data.
- Know how to compare or test data for differences and relative magnitude, and take appropriate action.
- Understand how the program flow can be diverted, based on the state of any bit or a zero value in a Data file.
- Recognize how the binary structure of the instruction word impacts on the usage of instructions.

Virtually all instructions act on data; either outside in its Data or Program memory space, or inside in an internal CPU register. Thus the 14-bit instruction code must include bits which inform the CPU's instruction decoder where this data is being held. The exception to this are the few Inherent instructions, such as `nop` (No Operation) and `return` (RETURN from subroutine). Before looking at the instruction set we will discuss the various techniques used to specify the location of any operands.

The general symbolic form of an instruction is:

```
instruction mnemonic <operand A>,<operand B>
```

where operand A is the source data or its location and operand B the destination. For example `movf 20h,w` (MOVe File) which copies data source out of File $20h$ to its destination in the Working register.

There are some variations on this structure. $2\frac{1}{2}$-operand instructions are common. For example, `addwf FILE,d` adds the `W` register's contents to the specified file's contents and deposits the result either in `W` or back in the file register itself. Thus `addwf 20h,f` means "add the contents of `W` to that of File $20h$ and put the outcome in File $20h$" or in Register Transfer Language (rtl, see page 49) `(f20) <- W + (f20)`. Of course this is not a true 3-operand instruction as the destination must be one of the two source locations; that is `W` or File $20h$. A few instructions have only a destination specified; for example, `clrf 20h`, and the inherent instructions have no explicit operands.

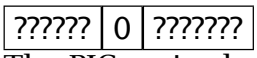
Instructions can be classified by their address mode.

Inherent

0000000	???????

The instructions listed in Appendix A, `clrwdt` (CLeaR WatchDog Timer), `retfie` (RETurn From Interrupt and Enable), `nop`, `return` and `sleep` do not explicitly refer to operands in memory or in the Working register. At the binary code level, all these instructions are coded with the upper seven bits zero. For example, `clrwdt` has a machine code of ***0000000***0000100b.

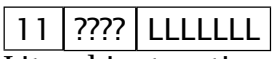
Register Direct



The PIC series has only one CPU register that can be explicitly specified by an instruction; the Working register. Where the *destination* is to be W then bit 7 of the instruction code is always 0. For example `clrw` is coded as 00000***0***0000011*b*. Many instructions can either use W or the source file as the destination, and this is coded by setting bit 7 to 0 or 1 respectively. In Appendix A, bit 7 is marked as d for destination (see also File Direct addressing) for applicable instructions.

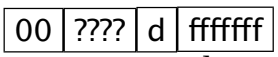
Where W is one of the source operands, the instruction normally shows this as part of the mnemonic. Thus `movwf f` copies W to the specified file register.

Literal



Literal instructions use the lower eight instruction word bits to specify a source operand which is *constant* data rather than data in a register. For example `addlw 06` is coded as ***11***1110***00000110***b*. The destination of this type of instruction is *always* the Working register, and this is shown in the mnemonic. Thus in our example, the sum W + 6 is copied back into W. In rtl this is expressed as `W <- W + #6` where the # (pound or hash) symbol denotes that the following number is a constant or literal rather than a file address.

File Direct



Instructions that specify that their source or destination operand lies in a file register use this address mode. The value of the file address is coded into the lower *seven* bits; denoted as fffffff. For instance the code for `clrf 20h` is ***00*** 0001 ***1 0100000***.

Most instructions altering the contents of a file register can 'dump' the outcome either in the Working register or else back in the file. Bit 7 of the instruction code, labelled d, can specify the destination as in the following example:

```
incf  20h,w  ; Coded as 00 1010 0 0100000
incf  20h,f  ; Coded as 00 1010 1 0100000
```

In both cases the contents of File 20*h* (File 01000000*b*) are incremented. In the former instance, the outcome is put in W leaving the file contents *unchanged* (d = 0), whilst in the latter the original data is *overwritten* (d = 1).

The main characteristic of this type of address mode is that the *location* of the operand is fixed as an integral part of the program, and thus cannot be altered as the program progresses. In some cases, such as in Program 5.1, this technique is rather inflexible.

As only seven bits of the instruction code are reserved for the file address, only files from 00 – 7F*h* may be directly accessed using this technique. However, from Fig. 4.6 on page 92 we see that the PIC16F84's Data store maps the register files in the range 00 – FF*h*, requiring an 8-bit address. The PIC16F84 gets round this by employing the RP0 bit in the Status register as a surrogate most-significant address bit – see Fig. 4.5 on page 89. This Register Page control bit can be altered, like any other read/write file register bit, to switch back and forth between Bank 0 (RP0 = 0) and Bank 1 (RP0 = 1).

The full 14-bit core CPU model has the capability of interacting with a 512-register file Data store. Devices with this size of Data store, such as the PIC16F87X line, have to deal with four banks of up to 128 register files. This requires a 9-bit address. Here the Status register, shown in Fig. 5.1, has two page select bits, RP1:RP0 which must be set up prior to using the File Direct address mode. For example, in such a Data store with a file address range 000 – 1FF*h*, in order to clear File 17F*h* (File 10 111 1111*b*) we need to set RP1:RP0 to 10:

```
bsf  STATUS,6    ; Make RP1 = 1
bcf  STATUS,5    ; Make RP0 = 0 (Bank 2)
clrf 17Fh        ; Clear File 17Fh
```

Program 15.4 on page 442 is an example making use of this extended bank switching.

File Indirect

00	????	d	0000000

Where data in the Data store is to be accessed, specifying its location directly as an address constant seems the obvious way to go; for example, `clrf 20h`. However, as we saw back in Program 3.1 on page 56, this may not always be the most efficient approach. This is especially the case when an array of data, such as a sequential set of readings, is to be processed.

Most MPU/MCU devices have one or more Address registers, sometimes known as Index registers. These are designed to hold the address in memory of the operand; that is they act as a **pointer**. Such processors use one or more Indexed or **Indirect** address modes which look to the appropriate pointer register to specify the operand location, rather than have the address as a fixed part of the instruction code. The advantage of this indirect approach of addressing a data operand is that the address can easily be altered as the program progresses. Thus, for example, an

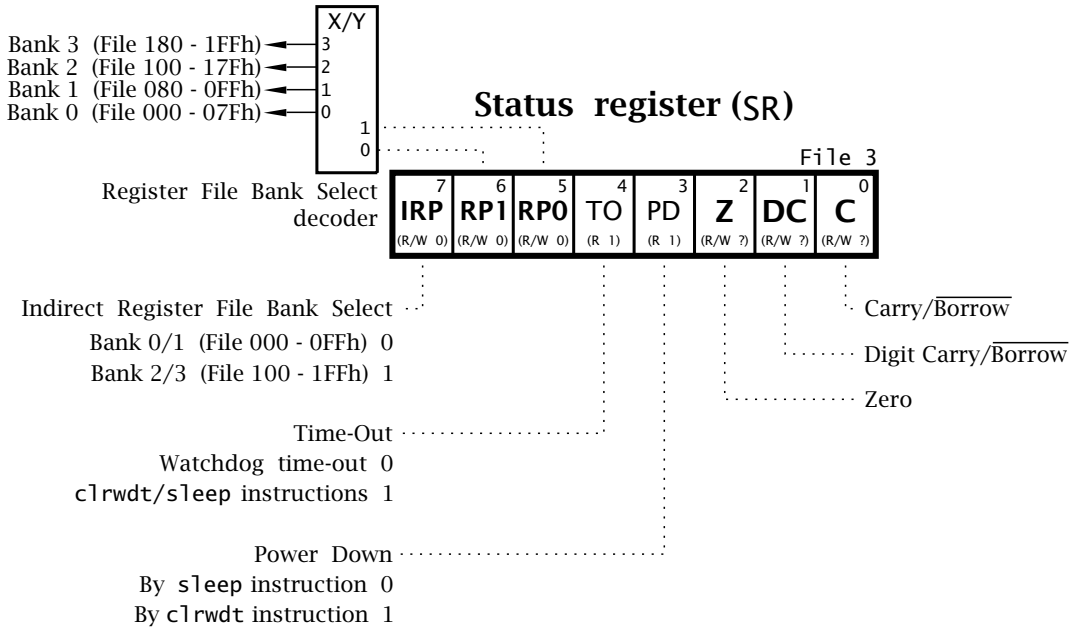


Fig. 5.1 General 14-bit core Status register.

array of data may be cleared by using an address register to point to the target location, and repeating in a loop while incrementing that pointer register.

The PIC family does not have dedicated CPU Address/Index registers to perform this indirect holding function. Instead, the pointer address is that contained in the **File Select Register** (**FSR**), which is File 4 in the Data store, see Fig. 4.6 on page 92. To activate the indirect mechanism, the normal File Direct address mode is used, but with File 0 as the target

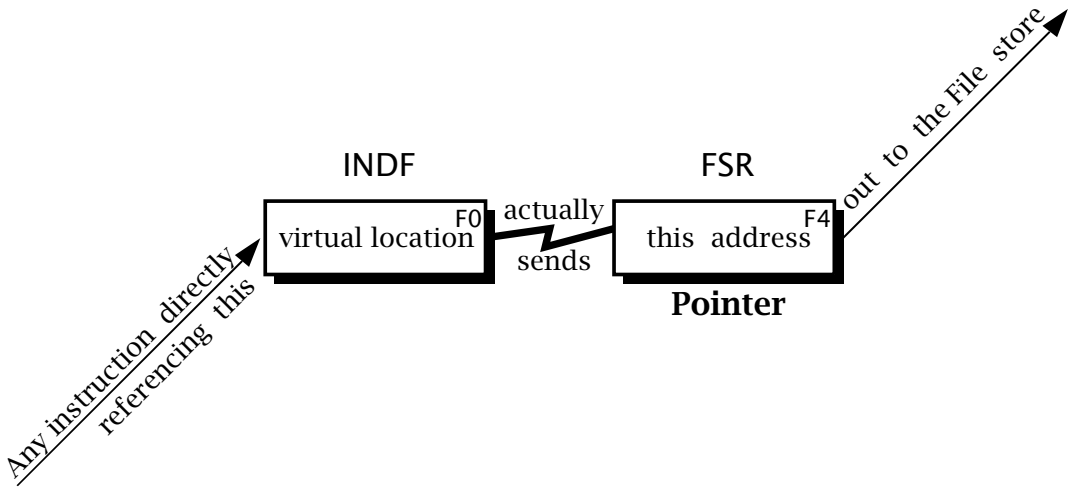


Fig. 5.2 The indirect mechanism.

location. File 0, the **INDF** (INDirect File) register, is a virtual location, that is it is not physically implemented. Its sole use is to trigger the use of the *contents of* the FSR as the operand address, as shown in Fig. 5.2. Thus the instruction `clrf 0` will actually clear the file whose 8-bit address is that in File 4. Of course the contents of the Special-Purpose Register (SPR) FSR can be altered at any time, for example incremented on each pass through a loop. This is the approach taken in Program 3.2 on page 57, which clears an array of Data-store memory. Although this approach to indirect addressing may seem rather convoluted, it does not require additional clock cycles to execute, unlike the alternative techniques used by other MPU/MCUs.

A more sophisticated example than that of Program 3.2 involves the sampling of temperature hourly over a daily period. With the assumption that the resulting array of 24 byte values are in situ in the Data store between File $30h$ and File $47h$, we are required to scan through the array looking for the maximum temperature. By the end of the routine this is to be in File $48h$.

To implement this procedure we first need a strategy or **task list**. One possibility would be:

1. Initialize Maximum as Temp[0].
2. IF Temp[1] > Maximum THEN Maximum = Temp[1].
3. IF Temp[2] > Maximum THEN Maximum = Temp[2].
4. IF Temp[3] > Maximum THEN Maximum = Temp[3].....
5. ...etc.
6. IF Temp[23] > Maximum THEN Maximum = Temp[23].
7. End.

How can we code Temp[i] > Maximum? If we subtract Maximum from Temp[i], that is Temp[i] − Maximum, then if a borrow is not generated (**C** flag set) we know that the former is higher than the latter and Maximum needs updated. Based on this, a possible coding is given in Program 5.1.

In this linear coding the **comparison** is implemented by first bringing the current maximum into the Working register (`movf MAXIMUM,w`), then subtracting it from the appropriate direct address, eg. for Temp[2] or File $32h$, `subwf TEMP_0+2,w`. This subtraction will set the **C** flag (that is $\overline{\mathbf{N}}$) if there is no borrow out, and in that situation the instruction `btfss` will skip to the update sequence. This simply copies down the appropriate temperature byte and copies it again up to the register file holding Maximum, eg. `movf TEMP_0+2,w` – `movwf MAXIMUM`.

The process outlined here has to be coded 24 times, with some small saving in the initial setting of Maximum to Temp[0] and the fact that this value is already in W for the Temp[1] comparison. This gives a total of 139 instructions. Execution time depends a little on the number of times Maximum has to be updated. However, taking a worse-case scenario and remembering that `goto` takes two cycles to implement as does `btfss`

Program 5.1 Finding the maximum temperature the linear way.

```
STATUS    equ   3           ; Status register is File 3
TEMP_0    equ   30h         ; Array starts @ File 30h
MAXIMUM   equ   48h         ; Maximum value to be in File 48h
NB        equ   0           ; Carry/Not Borrow flag is bit0

; Task1: Initialize Maximum as Temp[0]
MAX_DAILY movf  TEMP_0,w    ; Get it
          movwf MAXIMUM     ; and put it in as first maximum

; Task2: Check is Temp[1] > Maximum?
          subwf  TEMP_0+1,w  ; Temp[1] - Maximum
          btfss  STATUS,NB   ; IF no borrow (NB==1) THEN update
           goto  TASK3       ; Skip update
          movf   TEMP_0+1,w  ; Update by getting Temp[1]
          movwf  MAXIMUM     ; which is the new maximum

; Task3: Check is Temp[2] > Maximum?
TASK3     movf   MAXIMUM,w   ; Get current maximum
          subwf  TEMP_0+2,w  ; Temp[2] - Maximum
          btfss  STATUS,NB   ; IF no borrow (NB==1) THEN update
           goto  TASK4       ; Skip update
          movf   TEMP_2,w    ; Update by getting Temp[2]
          movwf  MAXIMUM     ; which is the new maximum

; Task4: Check is Temp[3] > Maximum?
TASK4     movf   MAXIMUM,w   ; Get current maximum
          subwf  TEMP_0+3,w  ; Temp[3] - Maximum
          btfss  STATUS,NB   ; IF no borrow (NB==1) THEN update
           goto  TASK5       ; Skip update
          movf   TEMP_0+3,w  ; Update by getting Temp[3]
          movwf  MAXIMUM     ; which is the new maximum

; Task5  and so on
TASK5     .....  ......

; Task24: Check is Temp[23] > Maximum?
TASK24    movf   MAXIMUM,w   ; Get current maximum
          subwf  TEMP_0+23,w ; Temp[23] - Maximum
          btfss  STATUS,NB   ; IF no borrow (NB==1) THEN update
           goto  FINI        ; Skip update
          movf   TEMP_0+23,w ; Update by getting Temp[23]
          movwf  MAXIMUM     ; which is the new maximum

FINI      .....  ......
```

when a skip occurs, then this gives 188 clock cycles to execute; that is 188 μs for a 4 MHz crystal.

Now Program 5.1 is rather a long program for a small task. This is because the complete sequence of compare-update instructions have to be implemented 23 times. Each sequence is identical, except the next ele-

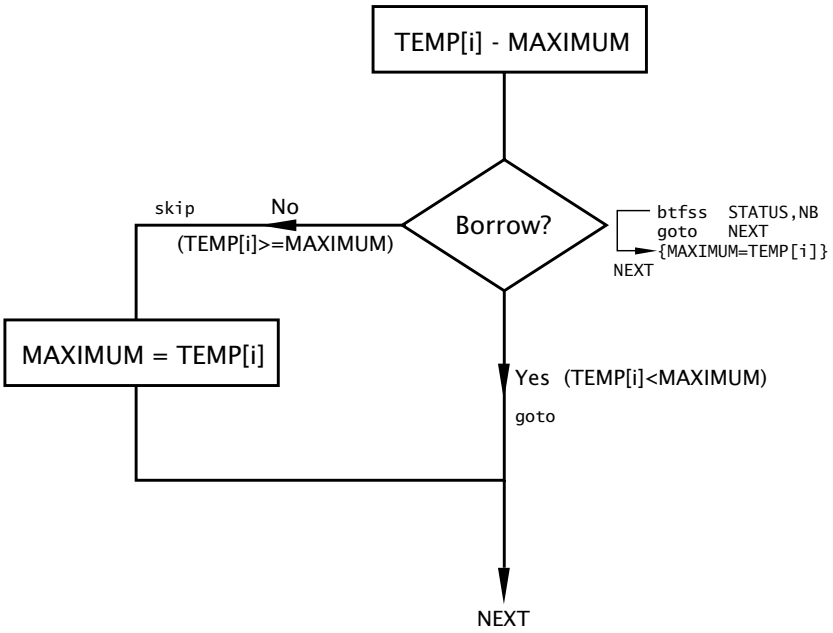


Fig. 5.3 The ith section of the compare-update sequence.

ment in the temperature array is tested each time. A much more efficient approach is to execute this sequence inside a loop and use an advancing pointer to target Temp[i] as the process unfolds. This technique leads to the task list:

1. Clear Maximum.
2. Point to Temp[0] (i = 0).
3. DO
   (a) IF Temp[i] > Maximum THEN Maximum = Temp[i].
   (b) Increment i.
   (c) Repeat WHILE i < 24.
4. End.

The implementation of Program 5.2 uses the same compare-update sequence, but this time with the Indirect address mode to access the data byte Temp[i]. The contents of the File Select Register here holds the address of Temp[i] and is initialized in Task 2. After each loop pass, this pointer is incremented and then compared by subtraction from the first address *beyond* the array; that is TEMP_0 + 24. If they are equal then the **Z** flag will be set and the `goto LOOP` instruction skipped over out of the loop.

This Indirect mode coding takes 14 instructions; that is 10% of the linear version. However, it does take rather longer to execute, due to the overhead of incrementing the pointer[1] and checking for range on each

[1]The PIC 17CXXX and 18CXXX series have auto incrementing and decrementing versions of Indirect addressing and more than one Indirect pointer register.

Program 5.2 Finding the maximum temperature using a loop structure.

```
INDF      equ   0           ; INDirect File register
STATUS    equ   3           ; Status register is File 3
FSR       equ   4           ; File Status Register
TEMP_0    equ   30h         ; Array starts @ File 30h
MAXIMUM   equ   48h         ; Maximum value to be in File 48h
Z         equ   2           ; Zero flag is bit2 of STATUS
NB        equ   0           ; Carry/Not Borrow flag is bit0

; Task1: Clear maximum
MAX_DAILY clrf  MAXIMUM

; Task2: Point to Temp[0]
          movlw TEMP_0      ; Put address of first temp byte
          movwf FSR         ; in the pointer register

; Task3:  DO
; Task3A: IF Temp[i] > Maximum THEN Maximum = Temp[i]
LOOP      movf   MAXIMUM,w  ; Get current maximum temperature
          subwf  INDF,w     ; Temp[i] - Maximum
          btfss  STATUS,NB  ; IF no borrow (NB==1) THEN update
           goto  NEXT       ; Skip update
          movf   INDF,w     ; Update by getting Temp[i]
          movwf  MAXIMUM    ; which is the new maximum

; Task3B: Increment i
NEXT      incf   FSR,f      ; i++

; Task3C: REPEAT WHILE i < 24
          movf   FSR,w      ; Get pointer address
          sublw  TEMP_0+18h ; Take away end address (Temp[24])
          btfss  STATUS,Z   ; IF equal THEN end
           goto  LOOP       ; ELSE repeat

; Task4: END
```

loop pass. The worst-case run time is 291 μs against 188 μs, assuming a 4 MHz crystal.

One advantage of the Indirect address mode is that the pointer address is eight bits wide. Thus it is not necessary to use the RP0 bit to switch between Bank 0 and Bank 1 of the Data store. The full 14-bit core model allows for four 128-byte banks of register files. The Status register of Fig. 5.1 shows the IRP (Indirect Register Page) bit which is used for family members with three or four register file banks. In such a device, if the temperature array of Program 5.2 were located in File $130h$ through File $147h$ then IRP would need to be set to 1 at the beginning of the routine (`bsf STATUS,7`) and cleared at the end. The rest of the code is unaltered. Microchip recommend that the IRP (and RP1) bits in

the Status register are left in their zero reset state in family members, such as the PIC16F84, that do not have data storage above Bank 1.

Bit

01	??	nnn	fffffff

Four instructions (as specified by the two ?? bits above) either alter or test the state of a single bit within a register file. In this situation the instruction word has an embedded 3-bit code `nnn` defining the bit number from 0 through 7, as well as the file address coded in the normal way. Thus the instruction `bcf 20h,7` (Bit Clear bit 7 in File 20*h*) is coded as ***01***00 ***111***0100000. The other instructions are `bsf` (Bit Set in File, coded as 01), `btfsc` (Bit Test File and Skip if Clear, coded as 10) and `btfss` (Bit Test File and Skip if Set, coded as 11).

Absolute

10	?	aaaaaaaaaaa

Two instructions allow the program to jump to another instruction anywhere in the Program store. These are `goto` and `call` (CALL or goto a subroutine, see Chapter 6). The 14-bit core allocates eleven bits of the instruction word to this absolute instruction address[2] in the Program store. Thus `goto 400h` would be coded as ***10***1 ***10000000000***. Similarly `call 530h` is ***10***0 ***10100110000***.

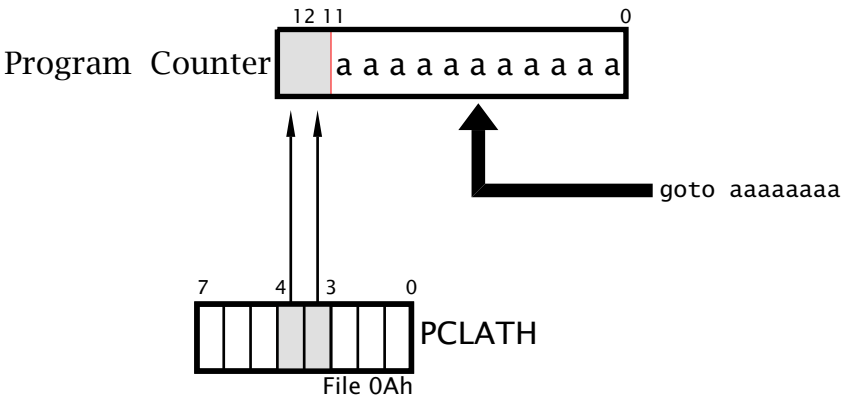


Fig. 5.4 Generating a 13-bit Program-store address for the `goto` and `call` instructions.

This 11-bit address can directly locate any instruction in a Program store of up to $2^{11} = 2$ Kbyte capacity. However, the mid-range core has

[2] Don't confuse this with the register file address in the Data store – in the Havard structure the two stores are logically distinct with different address spaces.

a 13-bit Program Counter which can potentially address a Program store of up to 8 Kbyte instructions; for example the PIC16C74 has a 4 Kbyte store. To cope with this situation, when a `goto` or `call` instruction is executed, the absolute 11-bit address is transferred into the PC together with bits 3:4 of the PCLATH (Program Counter LATch High) to make up an effective 13-bit Program-store address. This process is shown in Fig. 5.4 – see also Fig. 4.3 on page 86.

PCLATH is cleared on Reset, so the `goto` range is normally 000 – 7FF*h*. This covers all the address range for a 2 Kbyte store. For members with larger Program stores then a far `goto` and far `call` (i.e. beyond 7FF*h*) has to be implemented by twiddling bits PCLATH[4:3]. For example, in the PIC16C74 a `goto 800h` is coded as:

```
bsf  PCLATH,3      ; Make PCLATH(4:3) = 01
bcf  PCLATH,4
goto 800h          ; Go to it!
```

So far we have classified instructions by the method they pin-point their operands. The alternative approach is to catalog the instruction set by function. On this basis the 14-bit core PIC's instruction set can conveniently be divided into six groups, of which four will be examined here. Those relevant to subroutines and interrupts are listed in the next two chapters, and control instructions pertaining to internal operation of the MCU hardware are left to Chapter 10. The complete instruction set is given for reference in Appendix A on page 475.

Movement instructions

Around one in three instructions move data around without alteration. With this in mind the instructions in Table 5.1 will be the most used in the repertoire.

All three Move instructions can *copy* byte data to or from the Working register.

Table 5.1: Move instructions.

Operation	Mnemonic	Flags Z	DC	C	Description
Move					Copies a datum byte
Literal to W	`movlw k`	•	•	•	`[W] <- #kk`
File	`movf f,d`	√	•	•	`[d] <- [f]`
W to file	`movwf f`	•	•	•	`[f] <- [W]`
Swap					Interchanges file nybbles
File	`swapf f,d`	•	•	•	`[d] <- [F(3:0)][F(7:4)]`

•	Flag not affected	√	Flag operates in the normal way
W	Working register	f	File register
[ ]	Contents of	d	Destination, W or a file register
#kk	8-bit constant		

- **movlw** copies the specified 8-bit constant (or **literal**) to W. For example, movlw 80h initializes W to 10000000b. This instruction only affects the Working register and thus cannot be used directly to set up a file register to a constant value.
- **movwf** is used to copy out or **store** the contents of W into a register file. For example, the following code fragment will initialize the contents of File 22h to 80h:

```
movlw  80h   ; Set contents of W to 80h
movwf  22h   ; and copy to File 22h
```

- **movf** can copy (or **load**) the contents of any register file into W. For example, movf 22h,w loads W with the contents of File 22h.
  The destination of this instruction can also be the file itself, giving rise to seemingly useless instructions, such as movf 22h,f which copies the contents of File 22h back on top of itself! However, the process does activate the **Z** flag, which will be set if the file contents are zero, and of course this datum is not affected by the instruction. Thus movf FILE,f is equivalent to the missing tstf FILE instruction; that is TeST File for zero, that is commonly available in other MPU/MCUs. Thus the contents of any file register can be checked for zero by this means using a single instruction. An alternative technique needs to be used to test the contents of the Working register for zero.

Given that most instructions acting on a file can specify either the same file or the Working register as the destination, then a Move operation can be considered an implicit part of such instructions. As an example, for some situations to increment the contents of a file and then move it to W could be coded either as:

```
incf   22h,f  ; Increment File 22h's contents
movf   22h,w  ; and copy it into W
```

or

```
incf   22h,w  ; Copy the incremented File 22h's contents to W
```

Of course the latter does not actually change the state of the file.

The final instruction swapf swaps the top and bottom 4-bit nybbles in a file. Thus, for example, if File 22h was 1001 0111b then swapf 22h,f will yield 0111 1001b. If desired, the outcome destination could be specified as W, eg. swapf 22h,w. As swapf does not affect any flag, this latter form can be used as a transparent replacement for movf 22h,w which does alter the **Z** flag. Of course it does interchange the two nybbles in the process. Program 7.2 on page 183 shows this swap instruction used in this role.

Arithmetic

The PIC processors implement the normal byte-sized binary Add and Subtract instructions, as discussed on page 47, to add or subtract register

file contents to/from the Working register. In addition the W register may be added to or subtracted from an 8-bit constant.

Table 5.2: Arithmetic.

Operation		Mnemonic	Flags Z	DC	C	Description
Add						Binary addition
	Literal to W	`addlw k`	√	√	√	`[W] <- [W] + #kk`
	W to File	`addwf f,d`	√	√	√	`[d] <- [W] + [f]`
Clear						Zeroes destination byte or bit
	File	`clrf f`	√	•	•	`[f] <- #00`
	W	`clrw`	√	•	•	`[W] <- #00`
	Bit	`bcf f,n`	•	•	•	`[f_n] <- #0`
Decrement						Subtract one, produce no borrow
	File	`decf f,d`	√	•	•	`[f] <- [f] - #01`
Increment						Add one, produce no carry
	File	`incf f,d`	√	•	•	`[f] <- [f] + #01`
Set						Sets any bit in a file to one
	Bit	`bsf f,n`	•	•	•	`[f_n] <- #1`
Subtract						Binary subtraction
	W from literal	`sublw k`	√	√	√	`[W] <- #kk - [W]`
	W from File	`subwf f,d`	√	√	√	`[d] <- [f] - [W]`

`#0`	Single zero bit	`#1`	Single one bit
`#00`	Zero byte	`#01`	Byte $01h$
`#kk`	8-bit constant	`n`	3-bit bit specifier 0 – 7
f_n	Bit n of file		

As an example that uses most of the instructions in Table 5.2 consider the problem of dividing the contents of File $24h$ by an 8-bit divisor in W. The simplest way of doing this is to continually subtract the divisor from the dividend, keeping a count until a borrow is generated. The residue left after this last subtraction is the remainder with one divisor subtraction to many. Thus adding the divisor once can restore the remainder if this is needed.

A possible implementation based on this approach is given in Program 5.3. Here the quotient is cleared before entering the loop, using the **`clrf`** instruction. The loop itself simply increments the quotient using the **`incf`** instruction and then subtracts W (the divisor) from File $24h$ – **`subwf 24h,f`**. Both Subtract instructions generate a *complement borrow out*, which is represented by the **C** flag in the Status register – labelled NB for Not Borrow in the program. Thus the loop is exited when the Carry flag is *clear* after the subtract, which represents a borrow out.

On leaving the loop, the contents of File $20h$ needs to be decremented, as the last subtract was one too many. Using the **`decf`** instruction allows this correction to be applied directly on the file register.

Program 5.3 Division by repetitive subtraction.

```
STATUS    equ   3       ; Status register is File 3
NB        equ   0       ; Carry/Not Borrow flag is bit0
QUOTIENT  equ   20h     ; Quotient is held in File 20h
REMAINDER equ   21h     ; The remainder is put here
DIVIDEND  equ   24h     ; The dividend is here at the start

DIV       clrf  QUOTIENT    ; Zero the loop count

LOOP      incf  QUOTIENT,f  ; Record one loop pass
          subwf DIVIDEND,f  ; DIVIDEND - DIVISOR
          btfsc STATUS,NB   ; IF a borrow (NB==0) THEN exit loop
           goto LOOP        ; ELSE do another subtract/count

          decf  QUOTIENT,f  ; Compensate for one inc too many
          addwf DIVIDEND,w  ; Add divisor to residue
          movwf REMAINDER   ; which gives the remainder
          ..... ......      ; Next routine
```

The remainder can be determined from the residue left in the original dividend file. This represents one divisor subtracted too many. Thus, **`addwf DIVIDEND,w`** cancels this last action and this remainder outcome, now in W, is copied into File $21h$.

The **`addlw`** instruction can be used to add an 8-bit constant to W. Subtraction can also be carried out with this instruction by adding the 2's complement of the literal subtrahend. For example. `addlw F9h` or `addlw -7` will effectively subtract seven from the contents of W. Thus if `[W]` was $88h$ before this operation then the state of W after is $81h$:

$$\begin{array}{rr} 1000\ 1000 & W = 88h \\ +\ 1111\ 1001 & -7 = F9h \\ \hline 1000\ 0001 & 81h \end{array}$$

Rather confusingly this is not the same as **`sublw 7`** as this subtracts W from 7, that is `[W] <- 7-[W]`.[3]

Although the arithmetic instructions act on byte operands, operations on word sizes of greater than 8-bit precision are possible with the help of the Carry/Not-Borrow flag. The process for the addition of two n-byte objects is given by the task list:

1. For = 0 to n − 1 DO
   (a) Clear SUM.
   (b) SUM[i] = NUM1[i] + NUM2[i].
   (c) IF Carry[i] = 1 THEN increment SUM[i+1].
   (d) Increment i.
2. End.

[3] This foible is a major cause of errors in programming, and you should think carefully before using this instruction.

Example 3.2 on page 64 gives a practical implementation of this algorithm. Multiple-precision subtraction is carried out in a similar manner, using the Not-Borrow flag. Example 5.4 implements a 16-bit − 8-bit subtraction.

Data in memory can be incremented or decremented apparently in situ, although in reality it is transferred from the Data store into a temporary register, incremented or decremented using the ALU and transferred back to the Data store – a type of **read-modify-write** action. However, it still takes only one bus cycle to implement.

These instructions are especially useful in counting passes through a loop, as in Program 5.3 where QUOTIENT is located in the Data store at File 20*h*. However, `incf` is not quite the same as a `addlf 1,20h` type of instruction as it does not alter the state of the Carry flag. Thus if you wanted to increment a 32-bit number in Data memory at File 22:3:4:5*h* then this is how you would have to do it:

```
QP_INC incf  22h,f    ; Increment byte 1
       btfss STATUS,Z ; IF not overflowed to zero
        goto NEXT     ; THEN finished

       incf  23h,f    ; Increment byte 2
       btfss STATUS,Z ; again IF not overflowed to zero
        goto NEXT     ; THEN finished

       incf  24h,f    ; Increment byte 3
       btfss STATUS,Z ; IF not overflowed to zero
       incf  25h,f    ; increment byte 4

NEXT   ..... ......   ; Next code fragment
```

This depends on the algorithm *IF when byte n is incremented it wraps around from* FF*h to zero THEN increment byte n* + 1. See Example 5.1 for a multiple-precision decrement routine.

One of the more important operations is the *comparison* of the magnitude of the two numbers. Mathematically this can be done by *subtracting* the datum (designated `[f]` for either a register file or a literal) from the contents of the Working register `[W]`. The outcome gives the actual magnitude difference between the operands, but in most cases it is sufficient to determine the relative magnitude of the quantities – eg. is W higher than the datum? This is determined by checking the state of the **C** and **Z** flags in the Status register.

Working register *higher than* datum .....No borrow, non-zero
Working register *equal to* datum .....Zero
Working register *lower than* datum .....Borrow, non-zero

In terms of our processor, the **C** flag represents the *complement* of the borrow after subtraction and the **Z** flag is set on a zero outcome. This gives:

[W] *Higher than or equal*	[f]	:	[W]−[f] gives no borrow; (C = 1).
[W] *Equal to*	[f]	:	[W]−[f] gives Zero; (Z = 1).
[W] *Lower than*	[f]	:	[W]−[f] gives a borrow; (C = 0).

Consider as an example a fuel tank with a capacity of 255 liters, with a sensor at the bottom of the tank indicating the remaining volume of fuel as a linear function of pressure. Assume that the sensor represents the capacity as a byte that can be accessed at Port B (see page 95), which we give the name FUEL. We wish to write a routine that will light an 'empty' light (at bit 0 at Port A) if the capacity is below 20 liters and ring an alarm buzzer (bit 1 at Port A) if below 5 liters. Both output peripherals are active on logic 0. This is how it could be coded:

```
STATUS  equ  3     ; File 3 is the Status register
C       equ  0     ; Bit0 is the Carry flag
Z       equ  2     ; and bit2 is the zero flag
FUEL    equ  6     ; File 6 is Port B
DISPLAY equ  5     ; File 5 is Port A
LAMP    equ  0     ; Bit0 of which is the warning lamp
BUZZER  equ  1     ; and bit1 is the buzzer

ALARM  movf  FUEL,w          ; Read fuel gauge into W
       addlw -5              ; W-5 to compare.  IF C==1 THEN
       btfss STATUS,C        ; no borrow & FUEL HIGHER OR SAME
       bcf   DISPLAY,BUZZER  ; and sound buzzer

       movf  FUEL,w          ; Get fuel gauge again into W
       addlw -14h            ; W-20 to compare.  IF C==1 THEN
       btfss STATUS,C        ; no borrow & FUEL HIGHER OR SAME
       bcf   DISPLAY,LAMP    ; and light lamp

NEXT:  .....   .....         ; Continue
```

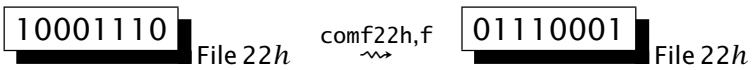
After each subtraction the Carry/$\overline{\text{borrow}}$ flag will be logic 1 (that is *no* borrow) if the datum in the Working register (the fuel reading) is higher or the same as the literal being subtracted – it is being compared with. The `addlw -k` instruction can be replaced by the more obvious `sublw k`.[4] Remembering that this subtracts W from the constant, that is k − W, then the following Skip on Set (`btfss`) instructions must be replaced by Skip on Clear (`btfsc`) to give the same sense of magnitude.

The contents of the Working register can be tested for zero in the same way, that is `addlw 0` or `sublw 0`. If W were zero then the outcome of this `tstw` type of instruction will set the **Z** flag. We have already seen that the instruction `movf xx,f` can be used in the same way as a `tst f` instruction to test File xx for zero. Note the use of the **`bcf`** (Bit Clear in File) instruction to clear the appropriate bit in Port A, which we assume to

[4]The Compare instruction of most MPU/MCUs is a subtract which sets the flags in the appropriate way, but which 'throws away' the difference outcome; that is does not overwrite the operand. A type of non-destructive subtract.

be initially set to output. In the same manner the **bsf** instruction could be used to turn off the lamp and buzzer at the beginning of the routine, as shown on page 125.

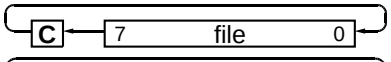
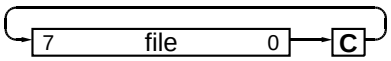
Logic and Shifting instructions
All four basic logic operations are provided, as shown in Table 5.3. The simplest of these is **comf** which inverts (or 1's COMplements) all bits in a file register. For example:



Alternatively the outcome can be placed in W with the original contents being unchanged; eg. `comf 22h,w`. There is no instruction `comw` to explicitly invert the contents of the Working register, but this can be accomplished in one bus cycle by subtracting from 11111111*b*; eg. `sublw 0FFh`: For example:

```
  11111111        Literal 0FFh
- 10001110        Working register
  --------
  01110001        1's complement of W
```

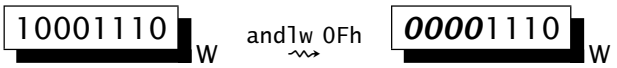
Table 5.3: Logic instructions.

Operation		Mnemonic	Flags Z	DC	C	Description
AND						Logic bitwise AND
	Literal to W	`andlw k`	√	•	•	`[W] <- [W] · #kk`
	W to File	`andwf f,d`	√	•	•	`[d] <- [W] · [f]`
Complement						Invert or NOT (1's complement)
	File	`comf f,d`	√	•	•	`[d] <-` $\overline{\texttt{[f]}}$
Inclusive-OR						Logic bitwise inclusive-OR
	Literal to W	`iorlw k`	√	•	•	`[W] <- [W] + #kk`
	W to File	`iorwf f,d`	√	•	•	`[d] <- [W] + [f]`
eXclusive-OR						Logic bitwise exclusive-OR
	Literal to W	`xorlw k`	√	•	•	`[W] <- [W] ⊕ #kk`
	W to File	`xorwf f,d`	√	•	•	`[d] <- [W] ⊕ [f]`
Rotate file shift						Circular shift into Carry
	Left	`rlf f,d`	•	•	b_7	
	Right	`rrf f,d`	•	•	b_0	

· Boolean bitwise AND
+ Boolean bitwise inclusive-OR
⊕ Boolean bitwise exclusive-OR
$\overline{\texttt{[f]}}$ Bitwise inverse of the file contents

The **andwf** instruction bitwise ANDs the Working register together with the contents of any file register, with the outcome being placed either in that same file or in W. Similarly the **andlw** instruction bitwise ANDs W with a byte literal.

ANDing an input with a 0 *always* gives a 0 output, whilst with a 1 does not change the logic value. For example:



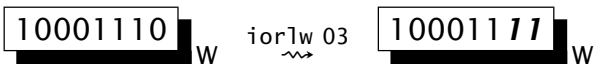
which zeros the upper nybble of the Working register. ANDing is normally used to clear any bit or bits in the destination operand. Thus `andlw b'00000011'`[5] clears the upper six bits in the Working register and leaves the lower two bits untouched. Of course, the `bcf` instruction can be used to clear a *single* bit in a file register.

Another use of ANDing is to check the state of any bit or bits in a datum; for example:

```
andlw   b'11000000'     ; Check bits 7 & 6
btfsc   STATUS,Z        ; IF not both zero THEN skip
 goto   ALL_ZERO        ; ELSE == 00, so go to ALL_ZERO routine
```

By ANDing the Working register with *11*000000*b*, the outcome will be all zero only if *both* bits 7 and 6 of W are 0. In this case the **Z** flag will be set and the following Bit Test File Skip on Clear will not be taken and the program will transfer to ALL_ZERO. If a *single* bit in a file register is being tested for zero then the `btfsc` instruction (see Table 5.4) is a more efficient process.

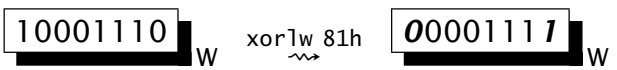
The **iorwf** and **iorlw** instructions implement the Inclusive-OR operation in the same way as for AND. ORing with a 0 leaves the source bit unchanged whereas ORing with a 1 sets the bit to a 1 irrespectively. Thus ORing is normally used to set any group of bits in the destination operand. For example:



sets the two least significant bits to one.

The **xorwf** and **xorlw** instructions provide for the eXclusive-OR operation. You will recall from page 14 that XORing with a 0 leaves a data bit unchanged, whilst XORing with a 1 inverts (or toggles) that bit. Thus, for example if we wished to invert both bits 0 and 7 of W:

For example:



Another use for XOR is to isolate *changes* between two bit patterns. From the truth table on page 14 we see that only when the two inputs *differ* is the output 1. Consider as an example a program routine that continually monitors Port B to which has been connected eight switches. This routine is waiting until someone moves a switch.

[5] Notice the assembler notation used to represent numbers in binary, see page 223.

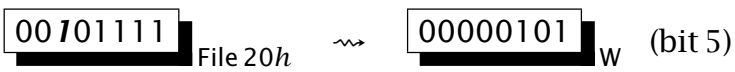
```
START  movf   PORTB,w    ; Get initial state of switches
       movwf  20h        ; Put away at File 20h

S_LOOP movf   PORTB,w    ; Sample switches
       xorwf  20h,w      ; Check for alterations
       btfsc  STATUS,Z   ; Skip out if check gives non zero
       goto   S_LOOP     ; ELSE check again
```

Two possible scenarios are:

The outcome in W reflects any changes. In the first case there are no differences between the latest sample and the original switch settings put away in File 20*h*. In the second situation Switch 4 has just been thrown from 1 to 0. You can determine which switch changed by shifting the outcome (the change byte) right, counting until the residue is zero. You can also determine the type of change (0 → 1 or 1 → 0) by ANDing the change byte to the original switch settings in File 20*h*, i.e. `andwf 20h,w`. If the outcome at bit 4 is a 0, then the change must have been 0 → 1, and vice versa.

The PIC 12/14-bit cores have two instructions which can shift a datum byte in a file register one place left or right. Both **`rlf`** and **`rrf`** are known as Circular or Rotate instructions. These shift left or right respectively with the incoming bit injected in from the Carry flag and outgoing bit popped out into the same **C** flag. This circular action is emphasized in Fig. 3.7 on page 61.[6]

One use of the shifting operation is to bitwise examine a datum. Suppose you want to determine the position of the leftmost logic 1 bit in File 20*h*, with this information being put in the Working register. For example, if the pattern is:



This can be realised by continually shifting the pattern under investigation right, counting the number of times until the residue is zero. The coding given in Program 5.4 uses the Working register as a counter. The data byte in File 20*h* is successively shifted right and the count incremented. As the Carry flag is cleared each time *before* the shift, logic 0s are brought in from the left.[7] Eventually the residue will become all zeros

[6]The PIC17CXXX series use the mnemonics `rlcf` and `rrcf` for Rotate Left/Right through Carry. The change allows for `rlnc` and `rrnc` for Rotate Left/Right Not through Carry.

[7]MPU/MCUs that have Logic Shift instructions always shift in 0s irrespective of the state of the **C** flag.

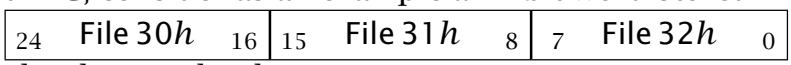
and the process terminated. Thus **0**0010111 (1) ⇝ **00**001011 (2) ⇝ **000**00101 (3) ⇝ **0000**0010 (4) ⇝ **00000**001 (5) ⇝ **000000**00.

Program 5.4 Shifting to find the highest set bit.

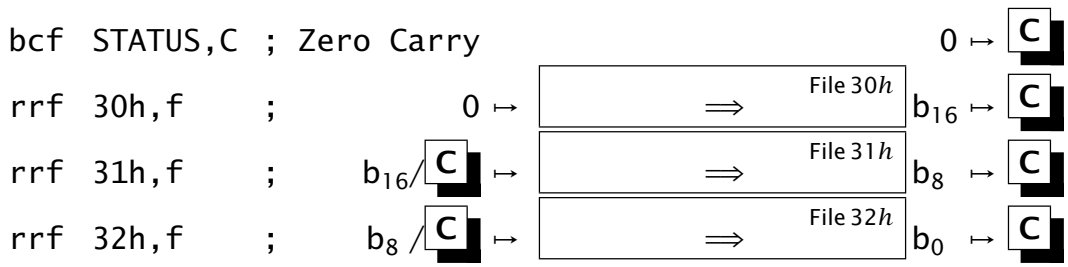
```
; Data is in File 20h, position of highest set bit to be in W
HIGH_BIT  clrw            ; Zero the count

; WHILE data is not zero, shift right and increment count
HLOOP     bcf   STATUS,C  ; Carry flag cleared
          rrf   20h,f     ; Shift rightmost bit into Carry
          btfsc STATUS,Z  ; IF not zero THEN continue
           goto FINI      ; ELSE exit

          addlw 1         ; Continue by adding one to count
           goto HLOOP     ; and do another shift
FINI      ..... ......
```

Shifting right pops out the rightmost bit into the Carry flag. Replacing `btfsc STATUS,Z` by `btfsc STATUS,C` would determine the position of the *rightmost* bit. In many situations repetitively shifting into the Carry flag can be used to examine the data on a bit by bit basis. For instance, we could modify our program to totalize the number of set bits in the byte, as in Program 5.5.

Program 5.4 did not distinguish between no bits set (00000000b) and bit 0 set (00000001b). How could you modify the program to do so?

The Rotate instructions can be used for multiple-precision shifting operations. Remembering that a Rotate takes in the Carry bit and in turn saves its ejected bit in **C**, consider as an example a 24-bit word stored in the data store at 

which can be shifted right once by the sequence:



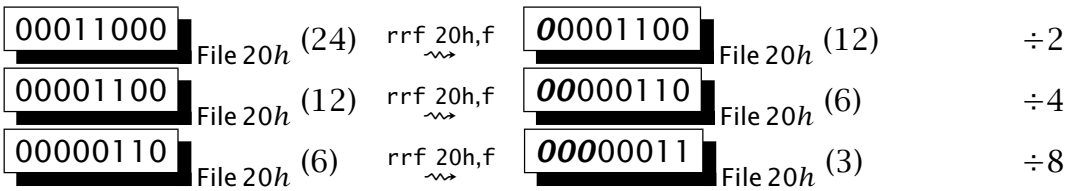
Consider that we wish to count the number of bits set to 1 in this triple-byte datum. One solution is shown in Program 5.5. Here the 24-bit word is shifted right (it could equally well have been left) until the word is all zero with the state of the Carry flag controlling the incrementation of the Working register. The multiple-precision zero test is implemented

Program 5.5 Triple-precision shifting to find the number of set bits.

```
COUNT_BIT  clrw             ; Zero the count
; WHILE data is not zero, shift right and increment counter
BLOOP      bcf   STATUS,C   ; Clear Carry flag
           rrf   30h,f      ; Shift right all three bytes
           rrf   31h,f
           rrf   32h,f
           btfsc STATUS,C   ; IF no carry then skip
           addlw 1          ; ELSE add one to count
; Now check for a triple-byte zero
           movf  32h,f      ; Test rightmost byte
           btfss STATUS,Z   ; IF zero THEN try middle byte
            goto BLOOP      ; ELSE shift again
           movf  31h,f      ; Test middle byte
           btfss STATUS,Z   ; IF zero THEN try leftmost byte
            goto BLOOP      ; ELSE shift again
           movf  30h,f      ; Test leftmost byte
           btfss STATUS,Z   ; IF zero THEN try middle byte
            goto BLOOP      ; ELSE shift again
EXIT       ..... ......     ; Exit with the count in W
```

by testing each byte in turn for zero and going back to the top of the loop if any byte test gives a non-zero outcome.

Shifting can be used to multiply and divide data by powers of two. For example, to divide by eight shift left three times:



where we are assuming that the Carry flag is cleared before each shift. In general shifting n places right gives a 2^n division and similarly to the left gives multiplication by the same factor – see page 11.

As an example consider that the byte in File 22h (called MULTIPLICAND) is to be multiplied by 3 to give a 2-byte product in File 24:5h (PRODUCT_H and PRODUCT_L respectively).

The implementation of Program 5.6 relies on factoring $\times 3$ as $\times 2 + \times 1$. The former is implemented by shifting a 16-bit extension of the byte multipicand (upper byte zeroed) once left. The single byte multiplicand is then added to the 2-byte subproduct to give the desired outcome. Example 5.5 gives the more complex case of multiplying by ten.

Program Counter instructions

The instructions listed in Table 5.4 modify in some way the setting of the Program Counter. The most elementary of these is **nop**. No OPer-

Program 5.6 Multiplying by three.

```
STATUS       equ  3   ; The Status register
C            equ  0   ; Bit0 of which is the Carry bit
MULTIPLICAND equ  22h ; File 22h is the multiplicand
PRODUCT_H    equ  24h ; File 24h is the High byte of the product
PRODUCT_L    equ  25h ; File 25h is the Low byte of the product

MULT_3  movf   MULTIPLICAND,w  ; Get xplicand from File 22h
        movwf  PRODUCT_L       ; and put as lower product byte
        clrf   PRODUCT_H       ; and extend to a 16-bit datum
; Now shift left 16 bits
        bcf    STATUS,C        ; Clear carry
        rlf    PRODUCT_L,f
        rlf    PRODUCT_H,f     ; giving x2
; Now add multiplicand (still in W) to give x2 + x1 = x3
        addwf  PRODUCT_L,f     ; Lower byte
        btfsc  STATUS,C        ; plus any carry
        incf   PRODUCT_H,f
        ....   ....
```

ation does not alter the state of the system in any way, but the PC will increment as a consequence of the instruction code being fetched from the Instruction store. Thus its sole outcome is [PC] <- [PC] + 1. This takes one bus cycle, so its main use is to implement a short delay, 1 μs for a 4 MHz clock rate. For example, to pulse Port A's pin low for 2 μs and then high we have:

```
    bcf  PORTA,0  ; Pin RA0 low
    nop           ; for 2 us
    nop
    bsf  PORTA,0  ; and now high
```

with the assumption that bit 0 of Port A has been set up as an output (see page 95) and that bit 0 (pin RA0) was high before entering the routine.

The **goto** instruction is an absolute jump instruction allowing the program to transfer to the specified instruction anywhere in the Program store. The process has been described in Fig. 5.4.

The remaining four instructions can skip over the *following* command if some condition is met. The pair **decfsz** (DECrement File and Skip on Zero) and **incfsz** (INCrement File and Skip on Zero) augment the specified file contents and then if the outcome is zero the PC is further incremented. Strangely, the **Z** flag is not affected by these instructions.

A typical use for these instructions is to count the number of passes through a loop. For example, suppose it is necessary to pulse Port A pin RA0 low 20 times.

Table 5.4: Program Counter instructions.

Operation	Mnemonic	Flags Z	DC	C	Description
Absolute jump					Goto a fixed instruction
Goto an instruction	`goto aaa`	•	•	•	`[PC] <- aaa`
No operation					Do nothing
	`nop`	•	•	•	`[PC] <- [PC] + 1`
Bit test and skip					Check bit in file and skip if true
Bit clear in File	`btfsc f,n`	•	•	•	`PC++ IF` f_n `== 0`
Bit set in File	`btfss f,n`	•	•	•	`PC++ IF` f_n `== 1`
Decrement and skip on zero					Decrement & skip if result is #00
File	`decfsz f,d`	•	•	•	`d <- f--, PC++ IF [f] == #00`
Increment and skip on zero					Increment & skip if result is #00
File	`incfsz f,d`	•	•	•	`d <- f++, PC++ IF [f] == #00`

`++` Increment contents `--` Decrement contents
`aaa` Absolute 11-bit instruction address

```
      movlw   d'20'     ; Put decimal 20 into W
      movwf   30h       ; & initialize File 30 as a loop counter
LOOP  bcf     PORTA,0   ; Pin RA0 low
      nop               ; for 2 us
      nop
      bsf     PORTA,0   ; and now high
      nop               ; for 2 us
      nop

      decfsz  30h,f     ; Count down
       goto   LOOP      ; Repeat loop if not zero
      .....   ......    ; ELSE escape the loop
```

Notice the assembler notation `d'20'` for *decimal* 20 – see page 223. This is equivalent to `14h` but more readily understood by the programmer.

The **`btfsc`** (Bit Test File and Skip if Clear) and **`btfss`** (Bit Test File and Skip if Set) instructions have been used extensively in programs both here and in Chapter 3. Besides their obvious use in changing the program flow based on the state of a specified bit in any register file, they allow decisions to be make on the state of the various flags in the Status register. Thus in Program 5.5 the series of `btfss STATUS,Z` (or the more unreadable `btfss 3,2`) instructions enable the program loop to be exited when the **Z** flag is set, that is on a series of zero outcomes. Similarly `btfsc STATUS,C` allows the count action to be skipped over when the **C** flag is clear. Neither instruction affects the flags.

All four skip instructions take two cycles to execute when the skip occurs but only one when the condition is not met. As described in Example 4.1 on page 99, the former situation needs to flush the instruction pipeline. This is to reflect the fact that the pre-fetched instruction code

sitting in stage one of the pipeline is not going to be the next instruction executed. This accounts for the additional bus cycle execution time as the true destination instruction has now to be fetched.

Examples

Example 5.1

Code a program to decrement a 2-byte number at File 26:27h ordered as high:low byte, remembering that `decf` does not alter the Carry/$\overline{\text{Borrow}}$ flag.

Solution

The task list to implement this job is:

1. IF the least significant byte in File 27h is zero THEN decrement the most significant byte.
2. Always decrement the least significant byte.

Program 5.7 gives one possible implementation based directly on this algorithm. Extension to an n-byte word is obvious.

Program 5.7 Double-precision decrement.

```
STATUS   equ  3          ; The Status register
Z        equ  2          ; Bit2 of which is the Zero bit
MSB      equ  26h        ; The Most Significant byte
LSB      equ  27h        ; The Least Significant Byte

         movf  LSB,f     ; Test for LSbyte zero.
         btfsc STATUS,Z  ; IF not THEN ship MSbyte decrement
         decf  MSB,f     ; ELSE must decrement MSByte
         decf  LSB,f     ; Always decrement LSbyte
```

Example 5.2

Some early computers used a bi-quinary code to represent BCD digits. This is a 7-bit code with only two bits set to one for any combination:

01	00001	0
01	00010	1
01	00100	2
01	01000	3
01	10000	4
10	00001	5
10	00010	6
10	00100	7
10	01000	8
10	10000	9

Although this is highly inefficient (with only ten out of a possible 128 code combinations being used as compared to the 16 combinations of the 4-bit natural code) it does have the advantage that it is very easy to determine when an error has occurred. Determine an error-detection routine to check the byte in File 20h. Assume that the most-significant bit is zero. If an error occurs then the Working register is to be set to FFh, otherwise zero.

Solution

All we need to do here is to determine when there are more or less than two bits set to one. Based on this approach we have the task list:

1. Count the number of ones in the bi-quinary byte.
2. Zero W.
3. IF count is not two THEN make FFh to signal an error.

Program 5.8 shows a possible coding implementing this algorithm. Here the loop continually shifts the bi-quinary byte left until the residue is zero. When the carry bit is set, the bit count is incremented. On exit from the loop, two is subtracted from the bit tally after moving into W. If it is zero then the routine is completed and the 00h setting of W shows a correct outcome. Otherwise FFh is placed in W to show an error situation. This is equivalent to decimal -1 and is traditionally used to note an error situation. As there are only ten legal combinations out of 128 possibilities used in this code the likelihood of an undetected error is rather small.

Example 5.3

Write a routine to convert a binary number of magnitude no greater than 63h (decimal 99) in File 20h to two BCD digits in File 21:2h ordered as Tens:Units – see page 6.

Solution

A possible algorithm to implement this binary to BCD conversion is to divide by ten; this generates a quotient between 0 and 9 (remember the

Program 5.8 Bi-quinary error detection.

```
STATUS     equ   3          ; Status register is File 3
C          equ   0          ; Carry flag is bit0
Z          equ   2          ; Zero flag is bit2
BI_QUIN    equ   20h        ; Bi-quinary byte is in File 20h
COUNT      equ   21h        ; The bit count is put here

BI_QUINARY clrf  COUNT      ; Bit count is cleared

LOOP       bcf   STATUS,C   ; Clear carry flag
           rlf   BI_QUIN,f  ; Rotate code left
           btfsc STATUS,C   ; IF no Carry popped out THEN skip
           incf  COUNT,f    ; the count increment
           movf  BI_QUIN,f  ; Test if residue is zero
           btfss STATUS,Z   ; IF zero THEN skip out of loop
            goto LOOP       ; ELSE repeat shift and count

           movf  COUNT,w    ; Get count
           sublw 2          ; Compare with two
           btfss STATUS,Z   ; IF ZERO finished with W == 00
           movlw 0FFh       ; ELSE put FFh (-1) in W
           ..... ......     ; and exit
```

maximum value is 99) and a remainder. The quotient is the number of tens and the remainder will be the number of units.

We have already seen how to divide by an arbitrary *variable* datum byte in Program 5.3. Here in Program 5.9 we wish to divide by a *constant* ten. This simplifies the coding somewhat with the `addlw -d'10'` (or `addlw -0Ah`) instruction used to take away the literal ten. Keeping a count in the `TENS` file register gives the number of subtractions until a borrow is generated. The required number of tens is one less than this tally; that is the number of successful subtractions. Adding that one extra ten back again to the residue gives the remainder, which is the units tally.

Example 5.4

Using Program 5.2 as a template write a program to evaluate the average temperature over the 24 hours.

Solution

Finding the average involves walking through the array adding each element to a 2-byte grand total. On completion divide by 24 to give the function:

$$\frac{\sum_{i=0}^{23} \mathsf{Temp}[i]}{24}$$

Program 5.9 Binary to 2-digit BCD conversion.

```
STATUS    equ   3       ; Status register is File 3
NB        equ   0       ; Carry/Not Borrow flag is bit0
BINARY    equ   20h     ; Binary byte is in File 20h
TENS      equ   21h     ; The quotient is put here
UNITS     equ   22h     ; The remainder is put here

; First divide by ten
BIN_2_BCD clrf  TENS          ; Zero the loop count
          movf  BINARY,w      ; Get binary byte into W

LOOP      incf  TENS,f        ; Record one ten subtracted
          addlw -d'10'        ; Subtract decimal ten
          btfsc STATUS,NB     ; IF a borrow (NB==0) THEN exit loop
           goto LOOP          ; ELSE do another subtract/count

          decf  TENS,f        ; Compensate for one inc too many
          addlw d'10'         ; Add ten to residue
          movwf UNITS         ; which gives the remainder
          ..... ......        ; Next routine
```

This gives the task list:

1. Clear AVERAGE.
2. Point to Temp[0] (i = 0).
3. DO
   (a) Add Temp[i] to the 2-byte grand total.
   (b) Increment i.
   (c) Repeat WHILE i < 24.
4. Divide by 24.
5. End.

Program 5.10 directly implements the task list, summing each datum byte by adding to the double-byte location File 48:9h, which has been cleared before entry to the loop. Division is accomplished by repetitively subtracting 24 from the final total. This is similar to the ÷10 routine of Program 5.9 but this time the single-byte constant is taken off the double-byte dividend. The number of successful subtracts is the quotient, which in this case is the truncated average. Of course it would be more accurate to round up if the remainder was more than half of the divisor.

Example 5.5

Write a routine to multiply a byte in File 22h by ten. The 2-byte product is to be located at File 23:4h giving $\boxed{\text{OVERFLOW}}_{\text{F 21}h}\boxed{\text{MULTIPLICAND}}_{\text{F 22}h} \times 10 = \boxed{\text{PRODUCT\_H}}_{\text{F 23}h}\boxed{\text{PRODUCT\_L}}_{\text{File 24}h}$ where File 21h is used to extend the single-byte multiplicand to a 16-bit double-byte datum.

Solution

The task list implemented in Program 5.11 splits the $\times 10$ task into a $\times 8 + \times 2$ operation. Thus:

1. Multiply multiplicand by eight (shift left three times).
2. Multiply multiplicand by two (shift left once).
3. Add the two 16-bit partial products.
4. End.

Program 5.10 Average daily temperature.

```
INDF      equ   0           ; INDirect File register
STATUS    equ   3           ; Status register is File 3
FSR       equ   4           ; File Status Register
TEMP_0    equ   30h         ; Array starts @ File 30h
SUM       equ   48h         ; Grand total to be in File 48:9h
AVERAGE   equ   4Ah         ; Average byte is to be here
Z         equ   2           ; Zero flag is bit2 of STATUS
C         equ   0           ; Carry flag is bit0
NB        equ   0           ; The alternative name is Not Borrow

; Task1: Clear grand total
AV_DAILY  clrf  SUM         ; MSbyte sum zeroed
          clrf  SUM+1       ; LSbyte sum zeroed

; Task2: Point to Temp[0]
          movlw TEMP_0      ; Put address of first temp byte
          movwf FSR         ; in the pointer register

; Task3:  DO
; Task3A: Add Temp[i] to the double-byte grand sum
LOOP1     movf  INDF,w      ; Get Temp[i]
          addwf SUM+1,f     ; Add LSB sum to it and put away
          btfsc STATUS,C    ; IF no carry, don't increment MSB
          incf  SUM,f       ; ELSE pass carry on

; Task3B: Increment i
NEXT      incf  FSR,f       ; i++

; Task3C: REPEAT WHILE i < 24
          movf  FSR,w       ; Get pointer address
          sublw TEMP_0+18h  ; Take away end address (Temp[24])
          btfss STATUS,Z    ; IF equal THEN end
          goto  LOOP1       ; ELSE repeat

; Task4: Divide by 24 to give the average
          clrf  AVERAGE     ; Zero the average

LOOP2     movlw d'24'       ; Put the constant 24 in W
          incf  AVERAGE,f   ; Record one subtract 24
          subwf SUM+1,f     ; Take away 24 from the sum LSB
          btfsc STATUS,NB   ; IF borrow out, goto high byte
          goto  LOOP2       ; ELSE do next subtract

          movlw 1           ; Subtract one from high byte
          subwf SUM,f
          btfsc STATUS,NB   ; IF a borrow (NB==0) THEN exit loop
          goto  LOOP2       ; ELSE do another subtract/count
          decf  AVERAGE,f   ; Compensate for one inc too many
          .....  ......     ; Next routine
```

Program 5.11 multiplication by ten.

```
STATUS       equ   3            ; Status register is File 3
OVERFLOW     equ   21h          ; Overflow to the multiplicand
MULTIPLICAND equ   22h          ; Multiplicand byte
PRODUCT_H    equ   23h          ; High byte of product
PRODUCT_L    equ   24h          ; Low byte of product
C            equ   0            ; Carry flag is bit0

; Task1: Multiply multiplicand by eight
MUL_10   movf   MULTIPLICAND,w ; Get Xcand byte
         movwf  PRODUCT_L      ; giving the lower product byte
         clrf   PRODUCT_H      ; extended to 16 bits

         bcf    STATUS,C       ; Clear Carry
         rlf    PRODUCT_L,f    ; Now shift word left
         rlf    PRODUCT_H,f    ; three times
         rlf    PRODUCT_L,f
         rlf    PRODUCT_H,f
         rlf    PRODUCT_L,f
         rlf    PRODUCT_H,f    ; Note Carry out here is zero
; Task2: Multiply multiplicand by two
         clrf   OVERFLOW       ; Extend Xcand to 16 bits
         rlf    MULTIPLICAND,f
         rlf    OVERFLOW,f

; Task3: Add X8 and X2
         movf   MULTIPLICAND,w ; Get LSB of X2
         addwf  PRODUCT_L,f    ; Add to LSB of X8
         btfsc  STATUS,C       ; Skip if no carry
         incf   OVERFLOW,f     ; ELSE add one onto MSByte
         movf   OVERFLOW,w     ; Get MSB of X2
         addwf  PRODUCT_H,f    ; Add to MSB of X8
         .....  ......         ; Next routine
```

The coding copies the 1-byte multiplicand into the lower byte of the product to be. Clearing the upper byte extends the datum to 16 bits. Clearing the Carry flag and then shifting left three times gives the $\times 8$ subproduct. As the upper byte is initially clear then the Carry flag is always zero after each double-byte shift, as it is going into the second $\times 2$ shift routine. This is done on the extended multiplicand memory space and the resulting subproduct added to first datum. This addition is simplified as the shifted upper byte of the $\times 2$ subproduct is small and so any carry from the lower byte addition is accounted for by simply incrementing the upper byte of this datum before the upper bytes are added. There will be no carry out from this latter addition.

Self-assessment questions

5.1 Can you deduce what function the following code fragment performs on the data byte in the Working register?

```
addwf  FILE,w
subwf  FILE,w
```

5.2 How could you extend Example 5.3 to give an outcome as packed BCD in File $21h$? Hint: Consider making use of the `swapf` instruction.

5.3 Develop Example 5.3 to give a 3-digit BCD outcome; removing the restriction that the original binary byte should be limited to decimal 99. The outcome is to be in File $21{:}2{:}3h$.

5.4 Extend Example 5.1 to decrement a quad-precision 32-bit word located at File $26{:}7{:}8{:}9h$, most significant byte first.

5.5 Example 5.4 evaluated the average of an array of hourly temperature samples by summing all bytes and then subtracting 24 until the residue dropped below zero. Write an extension to this program to round the average to the nearest integer; that is if the remainder is more than 12 then round up.

5.6 Write a routine to multiply a byte in File $22h$ by 13. The 2-byte product is to be located at File $23{:}4h$. The memory map for this is $\boxed{\text{OVERFLOW}}_{\text{F}21h}\boxed{\text{MULTIPLICAND}}_{\text{F}22h} \times 13 = \boxed{\text{PROD\_H}}_{\text{F}23h}\boxed{\text{PROD\_L}}_{\text{F}24h}$ where File $21h$ is used to extend the single-byte multiplicand to a 16-bit double byte datum. Note that this will require three shift and add processes.

5.7 A simple digital low-pass filter can be implemented using the algorithm:

$$\texttt{Array[i]} = \frac{S_n}{4} + \frac{S_{n-1}}{2} + \frac{S_{n-2}}{4}$$

where S_n is the nth sample from an eight-bit analog to digital converter located at Port B.

Write a routine assuming that the three byte memory locations to store S_n, S_{n-1} and S_{n-2} are located at File $20{:}1{:}2h$ respectively. The outcome `Array[i]` is to be located at File 48h.

5.8 A certain television show has eight contestants which are evenly divided into Team A and Team B. Each member has a switch, giving logic 1 when pressed, which may all be read simultaneously by the

microcontroller at Port B. Team A switches appear on the lower four bits of the port.

Write a routine that will:

- Decide when a response to the question has been made – any switch closed.
- Determine the team identity that has responded, by clearing File 20*h* for Team A and setting it to any non-zero value to signify Team B.
- Ascertain which team member pressed his or her switch by putting the member number 0–3 in File 21*h*.

CHAPTER 6

Subroutines and Modules

Good software should be configured as a set of interacting modules rather than one large program working straight through from beginning to end. There are many advantages to modular programming, which is almost mandatory when code lengths exceed a few hundred lines or when a project is being developed by a team.

What form should should such modules take? In order to answer this question we will look at the use of program structures designed to facilitate this modular approach and the instructions associated with it.

After completing this chapter you will:

- Appreciate the need for modular programming.
- Have an understanding of the structure of the stack and its use in the call–return subroutine and interrupt mechanisms.
- Understand the term nested subroutine.
- See how parameters can be passed to a subroutine, by copy or reference, and altered or returned to the caller.
- Be able to write a subroutine having a minimal impact on its environment.
- Be able to synthesize a software stack to open and close a frame in the Data store to pass parameters and provide a temporary workspace.

Take a look at the inside of your personal computer. It will probably look something like the photograph in Fig. 6.1, with a motherboard hosting the MPU, assorted memory and other support circuitry, and a variable number of expansion sockets. Into this will be plugged a disk controller card and a video card. There may be others, such as a soundboard, modem or network card. Each of these plug-in cards has a distinct and separate logical task and they interact via the services supplied by the main board – the motherboard.

There are many advantages to this **modular** construction.

- Flexibility; that is it is relatively easy to upgrade or reconfigure by adding or replacing plug-in cards.
- Can reuse from previous systems.
- Can buy in standard boards or design specialist boards in-house.
- Easy to maintain.

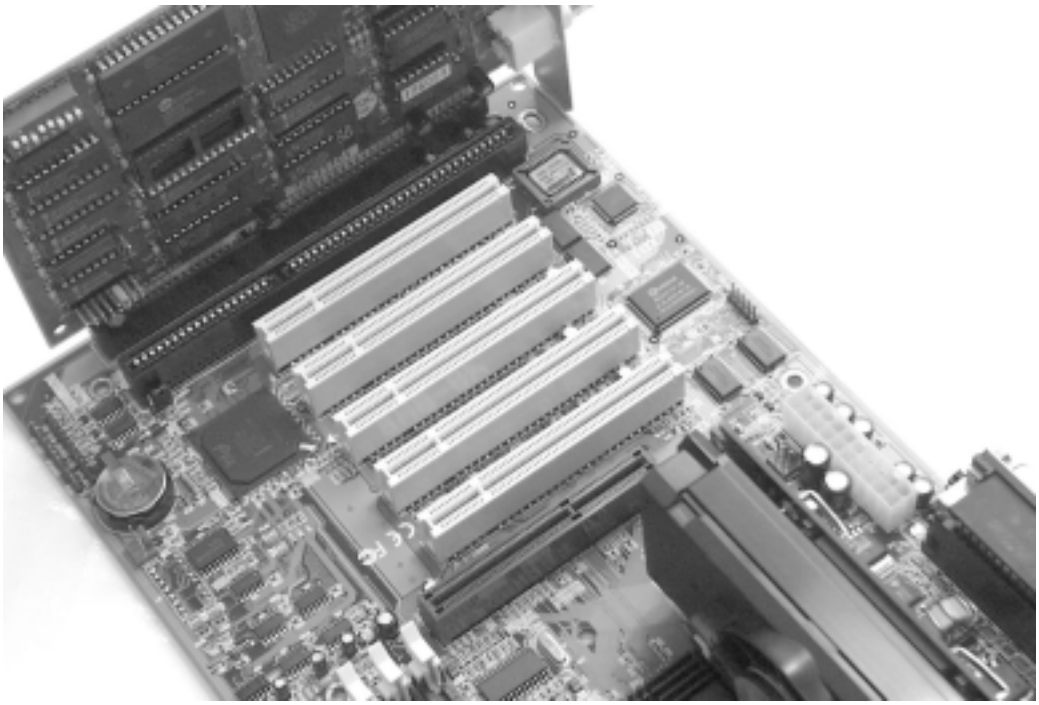


Fig. 6.1 Modular hardware implementing a PC.

Of course there are a few disadvantages. A fully integrated motherboard is smaller and potentially cheaper than an equivalent mother/daughterboard configuration. It is also likely to be more reliable, as input and output signals do not have to traverse sockets/plugs. However, when they do occur, faults are often more difficult to track down and rectify.

Modular programming uses the same principle to construct "software circuits", i.e. programs. A formal definition of modular programming[1] is:

> An approach to programming in which separate logical tasks are programmed separately and joined later.

Thus to write a program in a modular fashion we need to decompose the specification into a number of stand-alone routines, each implementing a well-defined task. Such a module should be relatively short, be well documented and easy for a human, not necessarily the original programmer, to understand.

The advantages of a modular program are similar to those for modular hardware, but even more compelling:

- Each module can be tested, debugged and maintained on a stand-alone basis. This makes for overall reliability.
- Can be reused from previous projects or bought in from outside.
- Easier to update by changing modules.

[1] From *Chambers Science and Technology Dictionary*, Cambridge University Press, 1988.

Deciding how to segment a program into individual stand-alone tasks is where the real expertise lies. The actual coding of such tasks as sub-programs is no different than the examples we have given in previous chapters, such as that shown in Program 5.11 on page 133. There are a few additional instructions associated with such sub-programs, and these are listed in Table 6.1. We will look at these and some useful techniques in constructing software in the remainder of the chapter.

Table 6.1: Subroutine and interrupt handling instructions.

Operation	Mnemonic	Description
Call		Transfer to subroutine
Call subroutine	`call aaa`	Push PC on to stack, PC <- <aaa>
Return		Transfer back to caller
from subroutine	`return`	Pull original PC back from Stack
	`retlw`	Put literal in W and return as above

Program modules may be entered by calling from other software or by a hardware event external to the processor. This may be a voltage at one of the processor pins or an internal peripheral interface wanting service, such as the Timer overflowing. In the former case modules at assembly level are universally known as **subroutines**, as they are in some high-level languages such as FORTRAN and BASIC.[2] In the latter they are classified as **interrupt service routines** or **interrupt handlers**. The techniques for writing these interrupt modules and their entry and exit techniques are sufficiently different to warrant a separate treatment in Chapter 7. Here we will look at subroutines.

Subroutines are the analog of hardware plug-in cards. Consider the situation where a 0.1 second (100 ms) delay task is to be implemented. This may be needed to alert an aircraft pilot to look at the control panel warning lights for various scenarios (such as low fuel or overheating) by sounding a buzzer for a short time. In a modular program, this delay would be implemented by coding a 100 ms subroutine which would be *called* by the main program as necessary. This is represented diagrammatically in Fig. 6.2.

In essence, calling up a subroutine involves nothing more than placing the address of the first subroutine instruction in the Program Counter (PC); that is doing a goto. Thus, if our delay subroutine were located at $400h$, then `goto 400h` would seem to do the trick. Assuming the programmer has labelled the subroutine entry point, usually the first instruction, `DELAY_100MS`, as in Program 6.1, then we have `goto DELAY_100MS`.

[2] Other high-level languages use the terms function (**C** and Pascal) or procedure (Pascal).

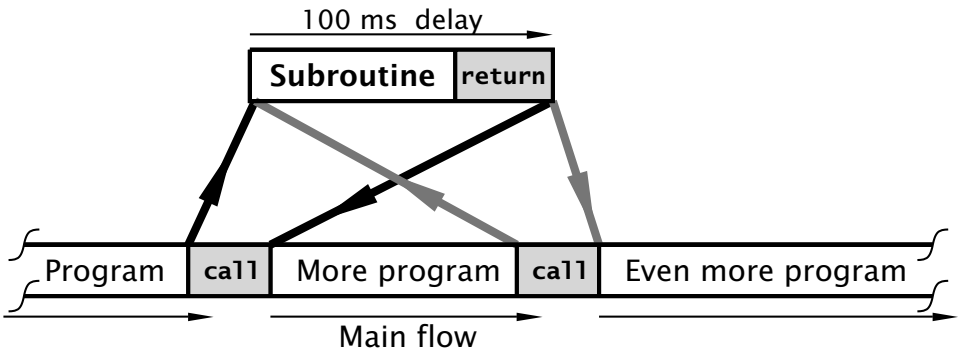


Fig. 6.2 Subroutine calling.

The problem really is how to get back again! Somehow the MPU has to remember from where in the caller program the subroutine was entered so that it can return to the *next* instruction in the caller's sequence. This can be seen in Fig. 6.2, where the jumping-off point can be from *anywhere* in the main program, or indeed from another subroutine – the latter process is called **nesting** – see Fig. 6.4.

One possibility is to place this address in a designated Address register or memory location prior to jumping off. This can then be moved back into the PC at the end of the subroutine as the return mechanism. This approach breaks down whenever one subroutine wishes to call another. Then the secondary subroutine will overwrite the return address of the first, and the main program can never be regained. To get around this problem, more than one register or memory location could be used to hold a stack of return addresses. This **last-in first-out stack** structure is shown in Fig. 6.3(a).

The 14-bit core PICs have a stack of eight 13-bit registers which are exclusively used to hold subroutine return addresses.[3] This structure, shown in Fig. 6.3, is known as a **hardware stack**. This stack is outside the PIC's normal memory map, so its contents cannot be altered by any normal process.[4]

Associated with this stack is a 3-bit counter which points to the next available register in the stack. This **Stack Pointer** (**SP**) cannot be explicitly altered by any instruction but it is *automatically* incremented each time a **call** instruction is executed. call is similar to a goto instruction, but before the specified instruction address is put into the Program Counter the current value of PC is pushed into the stack. This is the address of the instruction *after* the call instruction, as the PC has already been

[3] The 12-bit core devices have only two 11-bit stack registers and the 16-bit core PICs have a 16-deep stack.

[4] Most MPU/MCUs use an area of normal RAM together with a dedicated address register to implement their stack. This is much more flexible than a dedicated hardware stack but needs a more complex instruction set to manipulate the Stack pointer and to push and pull/pop data into and out of the stack.

incremented and the PIC is fetching this next instruction into the pipeline at the same time as the `call` instruction is being executed – see Fig. 4.4 on page 87.

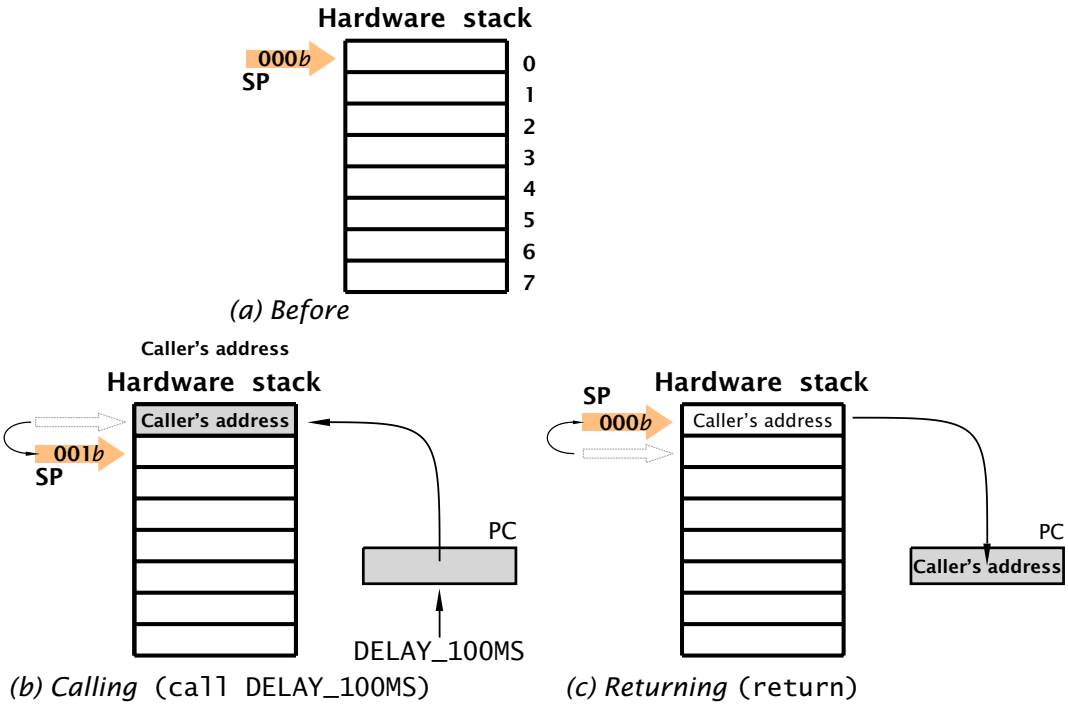


Fig. 6.3 Using the hardware stack hold return addresses.

In Fig. 6.3(b) the situation is shown after a call to a subroutine labelled `DELAY_100MS`. The execution sequence of this `call DELAY_100MS` is:

1. Copy the 13-bit contents of the PC into the stack register pointed to by the Stack Pointer. This will be the address of the instruction following the `call` instruction.
2. The Stack Pointer is decremented.
3. The destination address `DELAY_100MS`, that is the location of the entry point instruction of the subroutine, overwrites the original state of the PC. Effectively this causes the program execution to transfer to the subroutine.

Apart from the pushing of the return address into the stack in steps 1 and 2, `call` acts exactly like a plain `goto`. Thus `call` requires two bus cycles for execution as the pipeline needs to be flushed to remove the next caller instruction which is already in situ. This similarity also applies to the extension of its 11-bit absolute address in the `call` instruction word to a 13-bit Program store address using bits 3 & 4 of the PCLATH register, as shown in Fig. 5.4 on page 114. Far calls to subroutines located between

07FF*h* and 1FFF*h* are not required for the PIC16F84 and the majority of other 14-bit core PICs that have Program store capacities of not more than 2048 instructions.

The exit point from the subroutine should be a **`return`** instruction. This reverses the push action of `call` and pulls the return address back from the stack into the PC – as shown in Fig. 6.3(c). The execution sequence of `return` is:

1. Decrement the Stack Pointer.
2. Copy the address in the stack register pointed to by the Stack Pointer into the Program Counter.

Thus no matter whence the subroutine was called form it will return to the instruction just past the original `call` instruction when the subroutine has been completed.

`retlw` (RETurn Literal value in W)[5] is similar to the plain `return` instruction but places the specified constant byte in the Working register. Thus `retlw -1` could be used to return with W set to FF*h* (−1 decimal) to, say, indicate an error situation. Both Return instructions flush the pipeline and therefore take two bus cycles to execute.

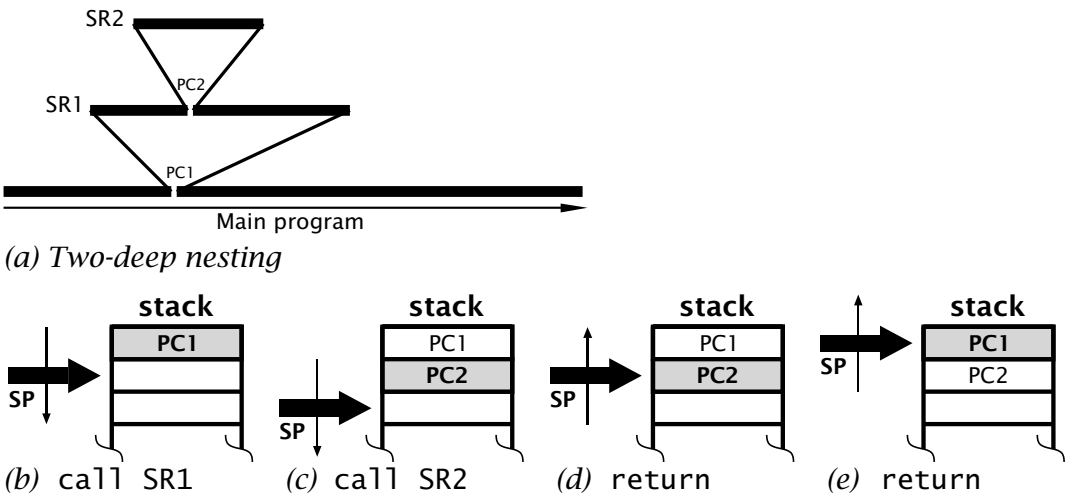


Fig. 6.4 Nested subroutines.

The beauty of the stack mechanism is its handling of **nested** subroutines. Consider the situation in Fig. 6.4 where the main program calls the first-level subroutine SR1 which in turn calls the second-level subroutine SR2. In order eventually to get back to the main program, the outward progression sequence must be exactly matched by the inward

[5] The 12-bit core PICs have only this `retlw` variant.

path. This pattern is matched by the **last-in first-out** (**LIFO**) structure of the stack mechanism, which can handle any arbitrary nesting sequence to any depth of eight subroutines automatically. It can even handle the (painful) situation where a subroutine calls itself! Such a subroutine is known as **recursive**. As we shall see in the next Chapter 7, the stack mechanism is also used to handle interrupts. Thus, in a system using both subroutines and interrupts the nesting depth will be somewhat less. The technique is so useful that virtually all MPU/MCUs support subroutines in this manner.

As the stack-Stack Pointer mechanism is part of the PICs hardware and requires no initialization, from the programmer's perspective only the following points are relevant:

- The subroutine should be invoked using the `call` instruction.
- The entry point to a subroutine should be labelled, and this label is then the name of that subroutine.
- The exit point from the subroutine should be either `return` or `retlw`, with the latter being used when a known constant is to be in the Working register on return – see Program 6.4.

As an example, let us code the 0.1 second (100 ms) delay subroutine of Fig. 6.2. Creating a delay in software is simply a matter of doing nothing for the appropriate duration. A common way of doing this is to count down an initial constant to zero. By choosing an appropriate constant, the delay can be tailored to the desired value. Obviously, this delay will depend on the PIC's oscillator rate. For the examples in this chapter we will assume a clock rate of 4 MHz, giving a bus cycle of 1 μs.

Consider the single-byte count code fragment:

```
D_LOOP  decfsz  COUNT,f    ; [(N-1)*1] + 2 cycles
         goto   D_LOOP     ; [(N-1)*2] cycles
```

This continually decrements the initial setting `N` of the register file labelled `COUNT`, dropping out when it reaches zero. The total delay is contributed by both instructions as:

1. The `decfsz` takes one cycle to execute, except when `N` reaches zero, in which case two cycles are needed as the pipeline needs to be flushed – see page 127. This gives a total execution time of $[(N-1)\times 1]+2$.
2. As the loop exits by skipping over the `goto` instruction, this is only executed $N-1$ times; each time taking two bus cycles. This contribution is thus $(N-1)\times 2$.

The total delay is thus:

$$\begin{aligned}[(N-1)\times 1]+2+(N-1)\times 2 &= [(N-1)\times 3]+2\\ &= (N\times 3)-1\end{aligned}$$

The maximum delay is when COUNT is initialized to zero which gives an effective value for N of 256.[6] In this situation the number of cycles in this code fragment is 767, plus the few cycles calling the subroutine (2˜), clearing COUNT (1˜) and returning from the subroutine (2˜), giving a total of 772˜. With a clock rate of 4 MHz, each cycle takes 1 μs (see Fig. 4.4 on page 87) giving a total ceiling 772 μs delay.

Program 6.1 A 100 ms delay subroutine.

```
; ************************************************************
; * FUNCTION: Delays for around 100ms with a 4MHz crystal    *
; * ENTRY   : None                                           *
; * EXIT    : Flags and W altered; Files 30:1h zero          *
; ************************************************************
COUNT_H  equ    30h        ; 2-byte counter
COUNT_L  equ    31h        ; at File 30:1h
N        equ    d'130'     ; Delay parameter

DELAY_100MS
         movlw  N          ; Set up high count to 130, 1˜
         movwf  COUNT_H    ; 1˜
         clrf   COUNT_L    ; and low count to 256, 1˜
D_LOOP
         decfsz COUNT_L,f ; Decrement LS count to zero
          goto  D_LOOP     ; taking in all H*[(256*3)-1]˜
         decfsz COUNT_H,f ; Repeat for the MS byte
          goto  D_LOOP     ; taking in all (H*3)-1˜
FINI     return
```

Program 6.1 uses two register files to extend this count. Whenever COUNT_L has decremented to zero, COUNT_H is decremented. This inner loop takes $(256 \times 3) - 1$ cycles in the normal way and is repeated H times, where H is the initial setting of COUNT_H. Only when this second byte reaches zero is the outer loop exited and execution returned to the caller. This second count contributes $(H \times 3) - 1$ cycles to the total. We thus need to determine the unknown value of H to give a total of $100,000 - 7$ cycles, taking into account the `call`, `return` and the three setting up instructions.

$$\begin{aligned} H \times [(256 \times 3) - 1] + (H \times 3) - 1 &= 100,000 - 7 \\ 767H + 3H - 1 &= 99,993 \\ H &\approx 130 \end{aligned}$$

This gives an actual delay of 100.107 ms; an error of less than 0.11%. Each incremental change in H gives an alteration of $\pm 770\,\mu$s.

The maximum delay available with this structure is 197 ms; however, adding one or more `nop` instructions at the beginning of the inner loop

[6] If N is zero then it will decrement 00 → FF → FE → · · · 01 → 00 and exit.

will increase the total delay by $H \times 256$ cycles and the same technique used in the inner loop adds H cycles for fine tuning. Example 6.3 gives an example of a very long triple count delay subroutine.

Our 100 ms delay program is an example of a double-void subroutine, in that no parameters (cf. signals in our hardware analog) are sent to it and nothing is returned – just the side effect of a delay (and the alteration of two register files, W and some Status register flags). Most subroutines process parameters made available at entry time and provide data at return time.

As a simple example, consider the extension of Program 6.1 to give a delay of $K \times 100$ ms, where K is a byte parameter 'sent' by the caller. The system view of this function is shown in Fig. 6.5 as a single input signal of range 1– 256, with no output signal – that is with a void output. This diagram also documents the location of all **local variables** used internally by the subroutine. This latter attribute is useful in checking for multiple usage of a file register between different subroutines and callers. Notice the double line vertical borders commonly used in flow diagrams to denote modules or subroutines.

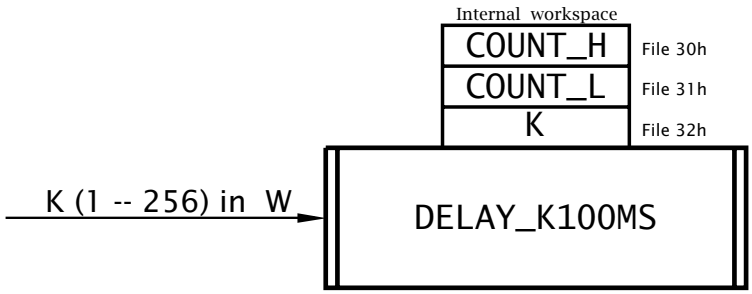


Fig. 6.5 System view of $K \times 100$ ms delay subroutine.

As there is only one input byte-sized parameter, the most convenient place to place K is in the Working register. Thus to call up a 5-second delay, the caller could use the sequence:

```
movlw  50             ; 50 x 0.1s gives 5 seconds
call   DELAY_K100MS   ; Go to it!
```

The actual subroutine itself in Program 6.2 implements the task list:

1. DO:
   (a) Delay 100ms.
   (b) Decrement K.
2. WHILE ($K > 0$).
3. End.

The actual coding simply copies the parameter from W into File $32h$ before entering the following delineated coding, which is identical to Program 6.1 and gives a single 100 ms delay. On completion of the delay K

Program 6.2 A $K \times 100$ ms delay subroutine.

```
; ************************************************************
; * FUNCTION: Delays for around K x 100 ms @ 4MHz            *
; * EXAMPLE : K = 100, delays 10 seconds                     *
; * ENTRY   : K in W, range 1 - 256                          *
; * EXIT    : Flags and W altered; Files 30:1:2h zero        *
; ************************************************************
COUNT_H  equ    30h       ; 2-byte counter
COUNT_L  equ    31h       ; at File 20:1h
K        equ    32h       ; Temporary storage for K
N        equ    d'130'    ; Delay parameter

DELAY_K100MS
         movwf  K         ; Put K away in a register file

; Task 1: DO 100ms delay
         movlw  N         ; Set up high count to 130
         movwf  COUNT_H
         clrf   COUNT_L   ; and low count to 256
DK_LOOP
         decfsz COUNT_L,f ; Decrement LS count to zero
          goto  DK_LOOP   ; taking in all H*[(256*3)-1]~
         decfsz COUNT_H,f ; Repeat for the MS byte
          goto  DK_LOOP   ; taking in all (H*3)-1~

; Task 2: Decrement K

         decfsz K,f

; Task 3: WHILE K > 0
          goto  DK_LOOP   ; REPEAT WHILE K > 0

FINI     return
```

is decremented in situ and the delay block repeated until K reaches zero. Thus the 100 ms block is repeated K times.

As K is tested for zero *after* the 100 ms delay is executed[7] an initial value of $K = 0$ will be treated as $K = 256$, giving a delay range of 00.1–25.6 s. Testing *before* the loop[8] would give a range 0–25.5 s. Actually, the delay will be a few μs longer than the plain 100 ms delay subroutine, due to the three additional instructions outside the delineated code block.

As W is needed to set up COUNT_H it could not be used directly to hold K during the subroutine. In fact, if the caller had known that File $32h$ was used by the subroutine to hold K then it could have been passed directly through this register file. However, the less the caller has to know about the 'innards' of its subroutines the better it will be, on the basis

[7] Known to **C** programmers as a DO-WHILE loop.
[8] Known to **C** programmers as a WHILE loop.

that a subroutine should disturb its environment as little as possible. DELAY_K100MS is not very good in this respect, using three file registers for its internal use and altering the Working register. As an example of what could go wrong, Program 6.3 shows an implementation of the task list but calling the 100 ms block as the existing Program 6.1 subroutine; that is a nested subroutine. Here File 30*h* is used as a store for *K* oblivious to the fact the this register file is used by subroutine DELAY_100MS as one of its counters. The effect of this interaction is to make *K* zero on return from DELAY_100MS, which when decremented at Task 2 will always give a non-zero outcome. Thus the delay is infinite and the system locks up! Simply changing `K equ 30h` to `K equ 32h` fixes the problem; but if another member of the team with responsibility for the DELAY_100MS subroutine alters its internal storage map without communicating this to other team members then catastrophe may occur! Thus even though each subroutine could have been passed when tested on its own, certain combinations of calling sequences could cause failure. We will return to this problem later.

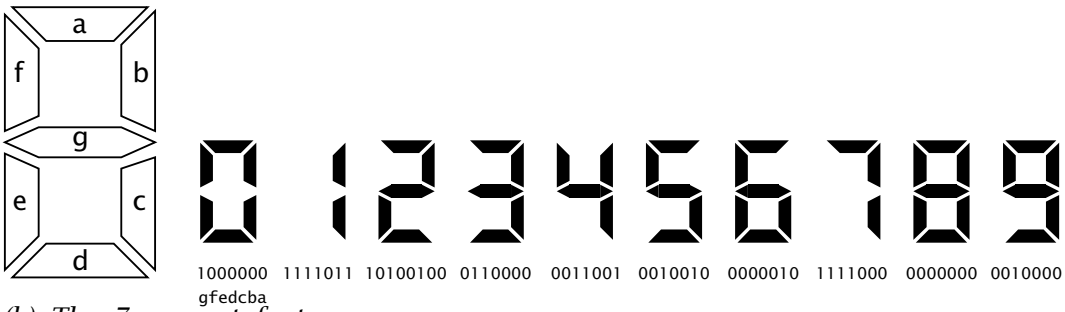
Program 6.2 is still void in that no data was returned to the caller on exit. For our next example we will code a subroutine that will activate a decimal readout. Many numeric electronic displays are based on a selective activation of seven segments in the manner shown in Fig. 6.6.

Program 6.3 An alternative $K \times 100$ ms delay subroutine.

```
; ************************************************************
; * FUNCTION: Delays for around K x 100 ms @ 4MHz            *
; * EXAMPLE : K = 100, delays 10 seconds                     *
; * RESOURCE: DELAY_100MS called                             *
; * ENTRY   : K in W, range 1 - 256                          *
; * EXIT    : Flags and W altered; Files 30:1h zero          *
; ************************************************************
K        equ    30h       ; Temporary storage for K

DELAY_K100MS
         movwf  K         ; Put K away in a register file

; Task 1: DO 100ms delay
DK_LOOP  call   DELAY_100MS

; Task 2: Decrement K

         decfsz K,f       ; Decrement K

; Task 3: WHILE K > 0
          goto  DK_LOOP   ; REPEAT WHILE K > 0

FINI     return
```

These segments are typically implemented using light-emitting diodes (see Fig. 11.13 on page 298) or electrodes in a liquid-crystal cell.

(a) System view



(b) The 7-segment font

Fig. 6.6 The 7-segment display.

The system description of our subroutine is shown in Fig. 6.6(a). Here the input signal is a 4-bit binary code representing the ten decimal digits as 0000 – 1001*b* in the Working register. The output, also in W, is the corresponding 7-segment code to activate the digit as listed in Table 6.2. This code assumes that a segment is lit/opaque on a binary 0 and unlit/clear on a binary 1.

Most MPU/MCUs deal with **look-up tables** by storing the codes as part of the program memory and copying the *N*th byte out of the table as the mapping function. In the 12- and 14-bit core PICs the Havard structure makes code in the Program store inaccessible to the program – but see Fig. 15.6 on page 445 for an exception. Instead, look-up tables are implemented as a series of **`retlw`** instructions, each returning a constant byte. This structure is shown in Table 6.2. As each `retlw` places an 8-bit code in W, I have arbitrarily made the unused bit 7 a logic 1.

In developing a coding based on this table structure, the mechanism for element *N* extraction is to execute the *N*th `retlw` instruction. This will place the instruction literal in the Working register and then do a normal return form subroutine back to the caller. In the example shown, if *N* is seven then the 7th `retlw` is executed returning with the code 11111000*b* for 8 in W.

The coding shown in Program 6.4 implements this selection mechanism by simply adding *N*, which is in W, to the lower byte of the Program Counter – that is PCL in File 2. PC then points to the *N*th `retlw` as desired.

Table 6.2: The 7-segment lookup table showing byte[N] being extracted.

	0	`retlw b'11000000'`	; 0
	1	`retlw b'11111001'`	; 1
	2	`retlw b'10100100'`	; 2
	3	`retlw b'10110000'`	; 3
	4	`retlw b'10011001'`	; 4
	5	`retlw b'10010010'`	; 5
	6	`retlw b'10000010'`	; 6
$N \Longrightarrow$	7	`retlw b'11111000'`	; 7 = Table[N]
	8	`retlw b'10000000'`	; 8
	9	`retlw b'10010000'`	; 9

Although this approach does work, there are limitations. Any alteration of the PCL register will cause both these eight bits together with the lowermost five bits of the PCLATH to be moved into the 13-bit PC – as described in Fig. 4.3 on page 86. This means that if the instruction `addwf PCL,f` causes the 8-bit PCL to overflow or if the contents of the PCLATH does not match the upper bits in the full PC, then the outcome stored into the PC will not be as the programmer wished – see Example 6.7. This is not easy to check, as the programmer is unlikely to know in advance where the subroutine is located in memory, that is what value the PC will have at the beginning of the subroutine. Even if he/she checks the assembler listing file (see Table 8.1 on page 206) for the value of SVN_SEG, this can change if subsequent alterations are made to other parts of the

Program 6.4 The software 7-segment decoder.

```
PCL     equ   2             ; Low byte of PC is at File 2

SVN_SEG addwf PCL,f         ; Add N to PCL giving PC + N
;               xgfedcba
        retlw b'11000000' ; Code for 0
        retlw b'11111001' ; Code for 1
        retlw b'10100100' ; Code for 2
        retlw b'10110000' ; Code for 3
        retlw b'10011001' ; Code for 4
        retlw b'10010010' ; Code for 5
        retlw b'10000010' ; Code for 6
        retlw b'11111000' ; Code for 7
        retlw b'10000000' ; Code for 8
        retlw b'10010000' ; Code for 9
```

program. It is possible to devise code to allow this address boundary to be crossed, but at the expense of complexity – see Example 6.7. The Microchip application note AN556 *Implementing a Table Read* gives techniques for dealing with these problems.

The code in Program 6.4 takes no account of the possibility that the datum in W is greater than $09h$. Of course it shouldn't be, but robust code should cope with all contingencies even if it is technically erroneous. This is especially true if the code module is to be reusable for general-purpose applications. What would happen if this situation arose and how could you add to the code to gracefully return an error code, say -1, in this eventuality?

Using W to transfer information to and fro a subroutine is limited to a single byte datum each way. Where several pieces of information of byte or greater sizes are to be passed, then file registers must be pressed into service for this conduit. An example of this is shown in Program 6.5 where two byte datums, labelled MULTIPLICAND and MULTIPLIER are to be multiplied giving a 16-bit outcome labelled PRODUCT_L:PRODUCT_H.

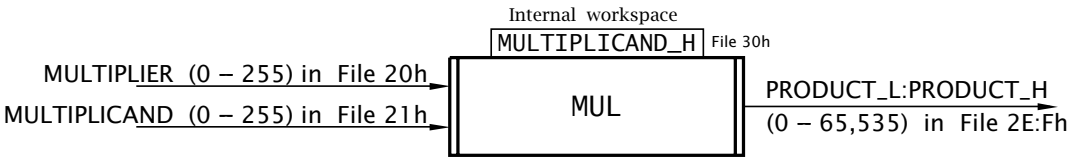


Fig. 6.7 System diagram for the byte multiplication subroutine.

The principle of the multiplication algorithm coded in Program 6.5 is a generalized version of that used by previous multiplication routines, such as Program 5.11 on page 133. Here the multiplier ten was decomposed to $\times 2 + \times 8$ which could be implemented by shifting once and three times respectively. In the more general case the multiplicand is shifted left and the nth shifted word added to the product if bit n of the multiplier is 1. Doing this eight times gives:

$$\text{Product} = \sum_{n=0}^{7} (\text{multiplicand} << n) \times \text{bit}\, n$$

where the << operator denotes shift left.

Using this **shift and add** algorithm gives the task list:

1. Zero double-byte product.
2. Extend Multiplicand to 16 bits.
3. DO
   (a) Shift Multiplier right once.

   (b) IF Carry bit is one THEN and Multiplicand to subproduct.
   (c) Shift Multiplicand right once.
   (d) Repeat WHILE product not zero.
4. End with 16-bit Product.

Program 6.5 declares the variables that are passed to and from the subroutine at the of the main program. Keeping all these global declarations in one part of the program and using a different file register for each overall global variable reduces the possibility of interaction but at the expense of rather extravagant use of scarce Data memory resources. Temporary local storage is declared within each subroutine as its need will be 'thrown away' after the subroutine is terminated. However, interaction can still occur in local storage where nested subroutine structures are used.

The coding follows the task list closely. The decision whether to add the left-shifted multiplicand to the subproduct is dependent on the state of the Carry flag when the multiplier is shifted right. This implements the conditional addition

$$\text{product} = \text{product} + (\text{multiplicand} << n) \times \text{bit}\, n$$

Rather than implementing this shift and conditional add process eights times, the summation loop is terminated whenever the multiplier residue is zero. This means that the execution time of the subroutine is variable, depending on the bit pattern of the multiplier. The worst-case scenario is when the multiplier is 255 ($11111111b$). This takes 142 cycles including the two cycle `call`.

In order to use this subroutine, the caller copies the multiplicand into File $20h$ and multiplier into File $21h$. On return, the 16-bit product can be read at File 2E:Fh. As an example, consider that the bytes located at File $42h$ and File $46h$ are to be multiplied.

```
movf  42h,w     ; Get Number 1
movwf 20h       ; and copy into MULTIPLIER
movf  46h,w     ; Get Number 2
movwf 21h       ; and copy into MULTIPLICAND
call  MULT      ; Go to it!
                ; On return the product is now in File 2E:Fh
```

Most MPU/MCUs have software stacks which in addition to saving subroutine return addresses allow the programmer to push and pull data to and from memory to pass information between caller and subroutine. As the stack is a dynamic storage entity, growing where necessary to accommodate these passed and temporary variables and shrinking again when the subroutine terminates, this clearly is an efficient method of memory allocation. Furthermore, each call outwards in a nested structure opens a new stack frame for this dynamic storage as an extension to the

Program 6.5 The byte multiplication subroutine.

```
; Global declarations
STATUS       equ   3             ; Status register is File 3
C            equ   0             ; Carry flag is bit0
Z            equ   2             ; and the Zero flag is bit2
MULTIPLIER   equ   20h           ; Multiplier byte
MULTIPLICAND equ   21h           ; Multiplicand byte
PRODUCT_L    equ   2Eh           ; Low byte of the product
PRODUCT_H    equ   2Fh           ; High byte of the product

; The MULT subroutine
; ************************************************************
; * FUNCTION: Multiplies two bytes to give a 2-byte product  *
; * EXAMPLE : MULTIPLICAND = 10h, MULTIPLIER = FFh           *
; * EXAMPLE : PRODUCT_H:PRODUCT_L = 0FF0h (16 x 255 = 4080d  *
; * ENTRY   : MULTIPLIER = File 20h, MULTIPLICAND = File 21h *
; * EXIT    : PRODUCT_H = File 2Eh, PRODUCT_L = 2Fh          *
; * EXIT    : MULTIPLIER, MULTIPLICAND altered               *
; * EXIT    : W, Status and MULTIPLICAND_H = File 30h altered*
; ************************************************************
; Local declarations
MULTIPLICAND_H equ 30h

; Task 1: Zero double-byte product
MUL      clrf   PRODUCT_L
         clrf   PRODUCT_H

; Task 2: Extend multiplicand to 16 bits
         clrf   MULTIPLICAND_H

; Task 3: DO
    ; Task 3A: Shift multiplier right once
MUL_LOOP bcf    STATUS,C         ; Clear carry
         rrf    MULTIPLIER,f

     ; Task 3B: IF Carry == 1 THEN add multiplicand to product
         btfss  STATUS,C         ; IF C == 1 THEN do addition
          goto  MUL_CONT         ; ELSE skip this task

         movf   MULTIPLICAND,w   ; DO addition
         addwf  PRODUCT_L,f      ; First the low bytes
         btfsc  STATUS,C         ; IF no carry THEN do high bytes
          incf  PRODUCT_H,f      ; ELSE add carry
         movf   MULTIPLICAND_H,w ; Next the high bytes
         addwf  PRODUCT_H,f

     ; Task 3C: Shift multiplicand right once
MUL_CONT bcf    STATUS,C         ; Zero Carry-in
         rlf    MULTIPLICAND,f
         rlf    MULTIPLICAND_H,f

     ; WHILE multiplier not zero
         movf MULTIPLIER,f       ; Test multiplier for zero
         btfss  STATUS,Z
          goto  MUL_LOOP         ; IF not THEN go again
         return                  ; ELSE finished
```

stack. in this way the possibility of overlap between variable storage when using nested subroutines is virtually eliminated.

High-level languages, such as **C** (see Chapter 9) are based around this stack model, which allows the creation and passing of variables only restricted by the amount of data memory that can be allocated to this stack.

The downside to this approach is the extra CPU resources necessary to support the creation and maintenance of the stack. One or more dedicated address registers or stack pointers are normally provided and address modes that facilitate access to variables in these stack frames are needed for efficient working. Even then, the outcome is normally slower and coding is longer than models based on fixed memory allocations.

The PIC CPU does not explicitly support a software stack. However, it is possible to simulate such a structure using Indirect addressing with the File Select Register (FSR = File 4) and INDirect File (INDF = File 0) – see page 109. As there is no stack pointer register per se, in the code fragment below the main routine has allocated File 0C*h* as a Pseudo Stack Pointer, which we call PSP.

```
; Global declarations

PSP           equ   0Ch   ; Holds the Pseudo Stack Pointer
TOS           equ   2Fh   ; File 2Fh is the initial Top Of Stack
INDF          equ   0     ; INDirect File
FSR           equ   04    ; File Select Register
STATUS        equ   3     ; Status register is File 3
C             equ   0     ; Carry flag is bit0
Z             equ   2     ; and the Zero flag is bit2
MULTIPLIER    equ   46h   ; Multiplier byte
MULTIPLICAND  equ   42h   ; Multiplicand byte

MAIN
; In the beginning set up Top Of Stack
 movlw    TOS
 movwf    PSP            ; which is stored in File 0Ch
;
;
; Sometime later when ready to call subroutine
 movf     PSP,w          ; Point FSR to top of stack frame
 movwf    FSR            ; which is held in the PSP

 movf     MULTIPLICAND,w ; Push multiplicand out into stack
 movwf    INDF
 decf     FSR,f

 movf     MULTIPLIER,w   ; Likewise for the multiplier
 movwf    INDF
 decf     FSR,f

 call     MUL            ; Go to it

; Continue on with the product available at FSR-3:FSR-4
```

The programmer also has to set aside a block of Data memory to hold the various stack frames. Here we are specifying that the Top Of Stack

(TOS) address is File 2F*h*. If the range File 2f*h*–0D*h* is kept clear of absolute allocations then a total of 35 bytes is available for the stack. As the hardware stack holds the subroutine return address, all locations in the simulated software stack can be used for variable passing and local storage. However, if an 8-deep subroutine nest is going to be implemented then a bigger slice of available storage may well be necessary. The software stack is initialized by moving the literal 2F*h*, named TOS, into the file holding the Pseudo Stack Pointer.

As an example, consider a stack-oriented version of the multiplication subroutine of Program 6.5. A view of the software stack from the perspective of this new coding is shown in Fig. 6.8. Based on this diagram, in order to call up this subroutine the following procedure has to be implemented:

1. Push the Multiplicand and then Multiplier into the stack frame and call the subroutine.
2. Push zero into the next byte in the frame, which is being used for local storage.
3. Push zero out twice more to create an initialized hole for the two bytes to return the product.

The following code fragment shows how item 1 above is coded.

(a) Transfer the contents of the Pseudo Stack Register to the FSR. This means that the FSR now points to the top of the new stack frame. If the subroutine is a first-level call (that is not nested from another subroutine) then this will be 2F*h* in our example.

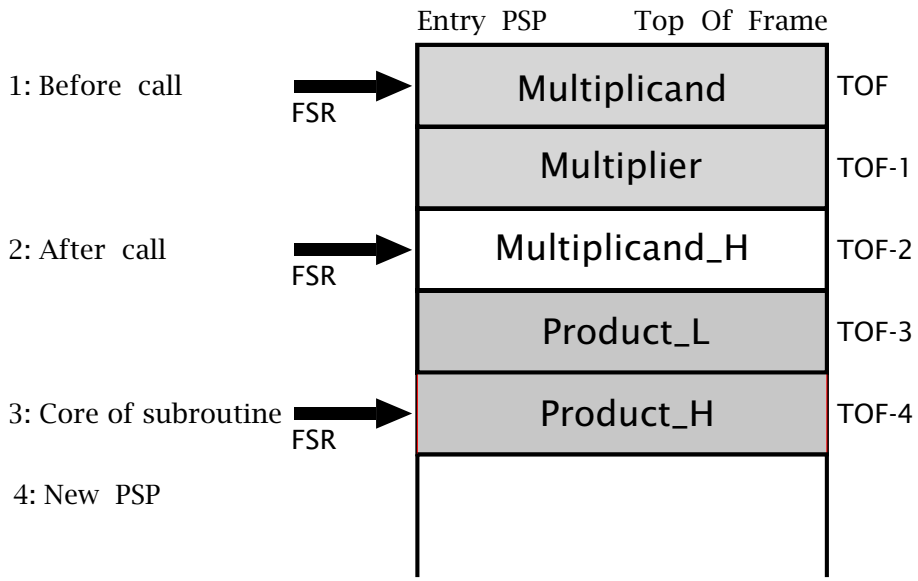


Fig. 6.8 The stack frame viewed from the perspective of subroutine MUL_S.

(b) Copy the Multiplicand from memory (we assume it is at File $46h$ as in our last example) into W and then indirectly into the frame using INDF as the target. Decrementing the FSR completes the push action.
(c) In a similar manner, the Multiplicand is pushed into the stack.
(d) Call the subroutine.

```
; (a)
movf  PSP,w           ; Copy current top of stack frame address
movwf FSR             ; into the File Select register

; (b)
movf  MULTIPLICAND,w  ; Push the Multiplicand into the stack
movwf INDF            ; by copying the datum out
decf  FSR,f           ; and decrementing the FSR

; (c)
movf  MULTIPLIER,w    ; Push the Multiplier into the stack
movwf INDF            ; by copying the datum out
decf  FSR,f           ; and decrementing the FSR

; (d)
call  MUL_S           ; Call the subroutine
```

Coding of the subroutine MUL_S is given in Program 6.6. This implements items 2–4 of Fig. 6.8. Firstly, MULTIPLICAND_H, the temporary storage of the multiplicand overflow, is zeroed and then zero is pushed into the next two locations to create the initial value for the product. In item 4 the Pseudo Stack Pointer is reset to point to the next free byte below the frame. In this way, should the subroutine wish to call another, then there will be a new frame available for that next-level storage with the new TOF beginning just below the old frame. These two instructions may be omitted in this case as there are no further nested call outs, although Task 3C will have to be altered. This is done in Example 6.6.

The core of the subroutine, that is Task 3, is similar to Program 6.5 but the FSR has to be moved up and down the frame to access the appropriate level. The only non-obvious use of the FSR is at Task 3C. As there are two ways into this routine, depending on whether the shifted multiplicand is added to the product or not, the state of the FSR is unknown. It can however be reset from the PSP which is pointing to just below the frame at this point. By adding five to this PSR value, the FSR will always point to MULTIPLICAND.

Finally the subroutine 'cleans up' the stack by updating the Pseudo Stack Pointer to its previous value. In this case this is done by adding five, but in general by adding the frame depth n.

Program 6.6 requires 45 instructions as compared to 20 in Program 6.5. Its worst-case execution time of 274 cycles also compares unfavorably

with 142 cycles. Thus in all respects except reusability and robustness this stack-based model is clearly inferior. It may be more economical in

Program 6.6 Implementing a byte multiply using a stack model. (continued next page).

```
; ************************************************************
; * FUNCTION: Multiplies two bytes to give a 2-byte product  *
; * EXAMPLE : MULTIPLICAND = 10h, MULTIPLIER = FFh           *
; * EXAMPLE : PRODUCT_H:PRODUCT_L = 0FF0h (16 x 255 = 4080d  *
; * ENTRY   : MULTIPLICAND = PSP, MULTIPLIER = PSP-1         *
; * ENTRY   : FSR points to one below MULTIPLIER             *
; * EXIT    : PRODUCT_H = PSP-3, PRODUCT_L = PSP-4           *
; * EXIT    : W, Status                                      *
; ************************************************************
; Tasks 1 & 2: Extend multiplicand and zero double-byte product
MUL      clrf   0
         decf   FSR,f     ; FSR ---> PRODUCT_L
         clrf   0
         decf   FSR,f     ; FSR ---> PRODUCT_H
         clrf   0
         decf   FSR,w     ; Now reset Pseudo Stack Pointer
         movwf  PSP       ; to Bottom Of Frame

; Task 3: DO
    ; Task 3A: Shift multiplier right once
         incf   FSR,f
         incf   FSR,f
         incf   FSR,f     ; FSR ---> MULTIPLIER
MUL_LOOP bcf    STATUS,C  ; Clear carry
         rrf    0,f

     ; Task 3B: IF Carry == 1 THEN add multiplicand to product
         btfss  STATUS,C  ; IF C == 1 THEN do addition
          goto  MUL_CONT  ; ELSE skip this task

         incf   FSR,f     ; FSR ---> MULTIPLICAND
         movf   0,w       ; DO addition
         decf   FSR,f
         decf   FSR,f
         decf   FSR,f     ; FSR ---> PRODUCT_L
         addwf  0,f       ; First the low bytes
         decf   FSR,f     ; FSR ---> PRODUCT_H
         btfsc  STATUS,C  ; IF no carry THEN do high bytes
          incf  0,f       ; ELSE add carry
         incf   FSR,f
         incf   FSR,f     ; FSR ---> MULTIPLICAND_H
         movf   0,w       ; Next the high bytes
         decf   FSR,f
         decf   FSR,f     ; FSR ---> PRODUCT_H
         addwf  0,f
```

Program 6.6 (continued.) Implementing a byte multiply using a stack model.

```
     ; Task 3C: Shift multiplicand right once
MUL_CONT movf   PSP,w     ; Reset FSR to the bottom of frame
         addlw  5
         movwf  FSR       ; FSR ---> MULTIPLICAND
         bcf    STATUS,C  ; Zero Carry-in
         rlf    0,f
         decf   FSR,f
         decf   FSR,f     ; FSR ---> MULTIPLICAND_H
         rlf    0,f

     ; Task 3D: WHILE multiplier not zero
         incf   FSR,f     ; FSR ---> MULTIPLIER
         movf   0,f       ; Test multiplier for zero
         btfss  STATUS,Z
          goto  MUL_LOOP  ; IF not THEN go again

     ; Task 4: End and clean up stack
         incf   FSR,f     ; FSR ---> Top Of Frame
         movf   FSR,w     ; Now reset Pseudo Stack Pointer
         movwf  PSP       ; To TOF
         return           ; Finished
```

its use of scarce data memory in large software systems. However, in programs running on low and mid-range PICs are often not very complex. Furthermore the small Program memory (1024 instructions for the PIC16F84) may further restrict the use of this relatively extravagant technique. Where real-time execution time is critical the additional burden of stack handling is unlikely to be worthwhile.

Examples

Example 6.1

The binary series approximation to the fraction $\frac{1}{3}$ is:

$$\frac{1}{3} = \frac{1}{2} - \frac{1}{4} + \frac{1}{8} - \frac{1}{16} + \frac{1}{32} - \frac{1}{64} + \frac{1}{128} \cdots$$

Using this series, write a subroutine that will divide a byte in the Working register by three with the quotient being returned in the same W register. The actual value of the summation up to $\frac{1}{128}$ is 0.3359375, which is within 0.78% of the exact value. With an 8-bit datum there is no point in including any further elements in the series, but where 16-bit operands are being processed then further elements up to the desired accuracy can be summed in the same manner.

Solution

The fractions $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$ etc. can be easily generated by shifting right. The coding listed in Program 6.7 simply repetitively shifts the number as copied to a temporary location in register file File 33*h* right. Notice that it is necessary to clear the Carry flag before each shift as both the previous Rotate and Subtract instructions can alter its state. As the subwf instruction subtracts the contents of W (the sub quotient) *from* the datum in the specified file register then the outcome being built up in W will oscillate in sign as desired. In situations where the series element signs

Program 6.7 Dividing by three

```
; ************************************************************
; * FUNCTION: Divides a 16-bit word by three                 *
; * EXAMPLE : Dividend = FFh (255d), Quotient = 55h (85d)    *
; * ENTRY   : Dividend N in W                                *
; * EXIT    : Quotient Q = N in W                            *
; * EXIT    : File 33h altered                               *
; ************************************************************

; Local declarations
DIV_TEMP equ 33h       ; Temporary work byte

DIV_3 movwf DIV_TEMP   ; Copy N into memory
      movlw 0          ; Clear quotient
      bcf   STATUS,C   ; Clear Carry flag

      rrf   DIV_TEMP,f ; N/2
      movf  DIV_TEMP,w ; Q = N(1/2)
      bcf   STATUS,C   ; Clear Carry flag
      rrf   DIV_TEMP,f ; N/4
      subwf DIV_TEMP,w ; Q = N/4-Q = N(+1/4-1/2)
      bcf   STATUS,C   ; Clear Carry flag
      rrf   DIV_TEMP,f ; N/8
      subwf DIV_TEMP,w ; Q = N/8-Q = N(1/8-1/4+1/2)
      bcf   STATUS,C   ; Clear Carry flag
      rrf   DIV_TEMP,f ; N/16
      subwf DIV_TEMP,w ; Q = N/16-Q = N(1/16-1/8+1/4-1/2)
      bcf   STATUS,C   ; Clear Carry flag
      rrf   DIV_TEMP,f ; N/32
      subwf DIV_TEMP,w ; Q = N/32-Q = N(1/32-1/16+1/8-1/4+1/2)
      bcf   STATUS,C   ; Clear Carry flag
      rrf   DIV_TEMP,f ; N/64
      subwf DIV_TEMP,w ; Q = N(1/64-1/32+1/16-1/8+1/4-1/2)
      bcf   STATUS,C   ; Clear Carry flag
      rrf   DIV_TEMP,f ; N/128
      subwf DIV_TEMP,w ; Q = N(1/128-1/64+1/32-1/16+1/8-1/4+1/2)
      return           ; Return with quotient in W
```

are not so regular then a further file register must be used to build up the quotient. The coding in Program 6.7 can be considered as the shift and subtract analog of the shift and add process outlined in Program 6.5 above. Execution takes 27 cycles including call and return. It can easily be extended to generate the fraction $\frac{2}{3}$ by omitting the first shift right, thereby effectively multiplying the series by two. The rest of the program remains the same.

Example 6.2
Write a subroutine to give a fixed 208 μs delay. Assume a 4 MHz processor clock rate.

Solution
For a short time period like this the code fragment of page 143 provides adequate delay. The solution shown in Program 6.8 is identical to this coding terminated by a `return` instruction.

Program 6.8 Coding a 208 μs delay.

```
COUNT      equ     34h      ; Temp location to hold count down
N          equ     d'67'    ; The delay parameter is decimal 67

DELAY_208  movlw   N        ; The delay parameter, 1~
           movwf   COUNT    ; Stored in File 34h , 1~

D_LOOP     decfsz  COUNT,f  ; [(N-1)*1] + 2 cycles
            goto   D_LOOP   ; [(N-1)*2] cycles

           return           ; Finish             , 2~
```

In order to calculate the parameter the time equation is:

$$\begin{aligned} 2(\texttt{call}) + 1 + 1 + (N \times 3 - 1) + 2 &= 208\,\mu s \\ N \times 3 &= 203 \\ N &= 67 \end{aligned}$$

This gives a total delay of 206 μs. Adding two `nop` instructions just before the `return` instruction will add the two extra μs.

See Program 12.10 on page 338.

Example 6.3
At the other end of the spectrum write a subroutine to give a delay of one second.

Solution

For a delay as long as this we need to extend Program 6.1 to use a larger count. In the coding of Program 6.9 three file registers are used to give a triple loop.

File register COUNT2 is initialized to the value H whilst the other two file registers are cleared giving an effective count range of 256. The outer loop exits the program when COUNT2 reaches zero. This single pass therefore contributes $(H \times 3) - 1$ cycles to the total delay. Each pass through the COUNT1 loop takes $(256 \times 3) - 1$ cycles and there are H passes giving a contribution of $H \times [(256 \times 3) - 1]$ cycles. In the same manner the inner loop based on decrementing COUNT0 runs $H \times 256$ times each pass contributing $(256 \times 3) - 1$ cycles to the total. Thus we have a total delay of:

$$(H \times 3) - 1 + H \times [(256 \times 3) - 1] + H \times 256 \times [(256 \times 3) - 1] + 6$$

Equating this to 10^6 (μs per second) gives $H \approx 5$. The actual delay with an H of 5 is 0.985615 s which is accurate to 1.4%. If desired the shortfall of 14,385 cycles can be made up by adding nop instructions to the middle count loop. Each such instruction gives an extra 1280 cycles. The

Program 6.9 A 1-second delay program.

```
COUNT0    equ     34h          ; 3-byte counter at F 34h
COUNT1    equ     35h          ; and F 35h
COUNT2    equ     36h          ; and F 36h
H         equ     5            ; The delay constant

; ************************************************************
; * FUNCTION: Delays for approx one second for a 4 MHz XTAL   *
; * ENTRY   : None                                           *
; * EXIT    : Status altered. W destroyed, Files 34:5:6h zero *
; ************************************************************
DELAY_1_S
        movlw   H            ; Put 5 as the MS count, 1~
        movwf   COUNT2       ; 1~
        clrf    COUNT1       ; Set lower counts to 256, 1~
        clrf    COUNT0       ; 1~
D_LOOP
        decfsz  COUNT0,f     ; Dec LSB count, H*256*[(256*3)-1]~
         goto   D_LOOP       ; to zero
        decfsz  COUNT1,f     ; Then dec NSB count, H*[(256*3)-1]~
         goto   D_LOOP       ; to zero and then repeat
        decfsz  COUNT2,f     ; Then dec MSB count, (H*3)-1~
         goto   D_LOOP       ; to zero and then repeat
        return               ; 2~
```

maximum delay possible with this program is 50.46 s for $H = 0$ (effectively 256).

Example 6.4

Design a subroutine to convert a binary byte passed in W to a 3-digit BCD equivalent in HUNDREDS (File 30h), TENS (File 31h) and UNITS (File 32h).

Solution

We have already coded a routine to implement this mapping in Example 5.3 on page 129. However this was restricted to a range 0–99, that is two digits. Nevertheless we can extend the technique used there by first subtracting and counting hundreds from the original binary byte. After this has been computed then the residue will be less than 100 and the rest of the coding will be the same, as shown in Program 6.10. Thus a suitable task list would be:

1. Divide by 100; the remainder is the hundreds digit.
2. Divide the quotient by ten; the remainder is the tens digit.
3. The quotient is the units digit.

Program 6.10 Binary to 3-digit BCD conversion.

```
; ************************************************************
; * FUNCTION: Converts a binary byte in W to three BCD digits*
; * EXAMPLE : Binary = FFh (255d), HUNDREDS = 02h            *
; * EXAMPLE : TENS = 02h, UNITS = 05h                        *
; * ENTRY   : Binary  in W                                   *
; * EXIT    : HUNDREDS = hundreds digits, TENS = tens digit  *
; * EXIT    : UNITS = units digit. W holds units             *
; ************************************************************

; First divide by a hundred
BIN_2_BCD  clrf  HUNDREDS    ; Zero the loop count

LOOP100    incf  HUNDREDS,f  ; Record one hundred subtracted
           addlw -d'100'     ; Subtract decimal hundred
           btfsc STATUS,NB   ; IF a borrow (NB==0) THEN exit loop
            goto LOOP100     ; ELSE do another subtract/count

           decf  HUNDREDS,f  ; Compensate for one inc too many
           addlw d'100'      ; Add a hundred to residue

; Next divide by ten
          clrf  TENS         ; Zero the loop count

LOOP10     incf  TENS,f      ; Record one ten subtracted
           addlw -d'10'      ; Subtract decimal ten
           btfsc STATUS,NB   ; IF a borrow (NB==0) THEN exit loop
            goto LOOP10      ; ELSE do another subtract/count

           decf  TENS,f      ; Compensate for one inc too many
           addlw d'10'       ; Add ten to residue
           movwf UNITS       ; which gives the remainder
           return            ; and return to caller
```

Example 6.5

Write a subroutine to evaluate the square root of a 16-bit integer located in File 26:7h. The 8-bit outcome is to be returned in the Working register.

Solution

The crudest way of doing this is to try every possible integer k from 1 upwards, generating k^2 by multiplication and checking that the outcome is no more than n. A slightly more sophisticated approach is based on the relationship:

$$k^2 = \sum_{i=0}^{k} (2 \times i) + 1$$

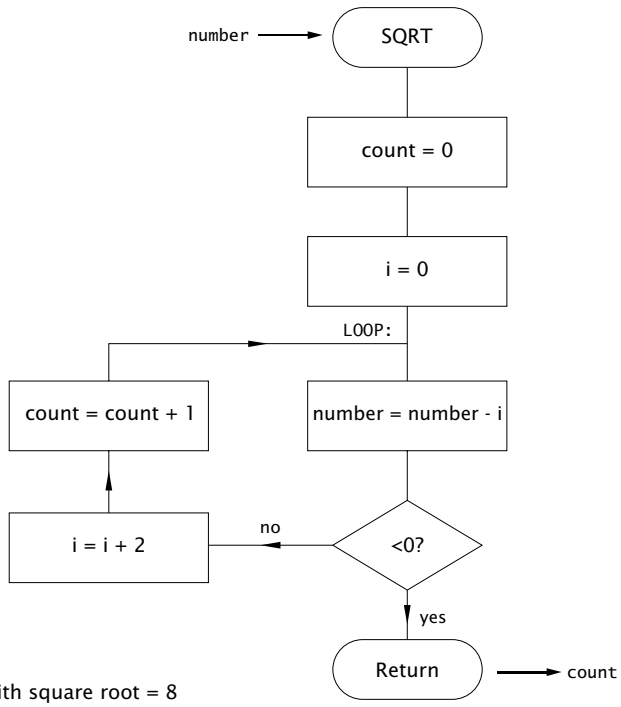
On this basis a possible structure for this function is:

1. Zero the loop count
2. Set variable I (the magic number) to 1
3. DO forever:
   (a) Take I from Number
   (b) IF the outcome is under zero THEN BREAK out
   (c) ELSE increment the loop count
   (d) Add 2 to I
4. Return loop count as $\sqrt{\text{Number}}$

That is sequentially subtract the series 1, 3, 5, 7, 9, 11…from Number until underflow occurs; with the tally of successful passes being the square

```
 65   count = 0
- 1
 64   count = 1
- 3
 61   count = 2
- 5
 56   count = 3
- 7
 49   count = 4
- 9
 40   count = 5
-11
 29   count = 6
-13
 16   count = 7
-15
  1   count = 8
-17
-16   Terminate with square root = 8
```

(a) An example



(b) Flowchart of the process

Fig. 6.9 Finding the square root of an integer.

root. An example giving $\sqrt{65} = 8$ is given in Fig. 6.9(a) using this series approach. A flowchart visualizing the task list is also given in Fig. 6.9(b).

Program 6.11 Coding the square root subroutine.

```
; Global declarations
STATUS       equ   3      ; Status register is File 3
C            equ   0      ; Carry flag is bit0
NB           equ   0      ; Alternative name Not Borrow
NUM_H        equ   26h    ; Number low byte
NUM_L        equ   27h    ; Number high byte

; ************************************************************
; * FUNCTION: Calculates the square root of a 16-bit integer *
; * EXAMPLE : Number = FFFFh (65,535d), Root = FFh (255d)    *
; * ENTRY   : Number in File 26:7h                           *
; * EXIT    : Root in W. Files 26:7h and 35:6:7h altered     *
; ************************************************************

; Local declarations
COUNT    equ    35h      ; The loop count
I_H      equ    36h      ; Magic number high
I_L      equ    37h      ; Magic number low

; Task 1: Zero loop count
SQR_ROOT clrf   COUNT

; Task 2: Set magic number I to one
         clrf   I_L
         clrf   I_H
         incf   I_L,f

; Task 3: DO
; Task 3(a): Number - I
SQR_LOOP movf   I_L,w     ; Get low byte magic number
         subwf  NUM_L,f   ; Subtract from low byte Number
         movf   I_H,w     ; Get high byte magic number
         btfss  STATUS,NB ; Skip if No Borrow out
          addlw 1         ; Return borrow
         subwf  NUM_H,f   ; Subtract high bytes

; Task 3(b): IF underflow THEN exit
         btfss  STATUS,NB ; IF No Borrow THEN continue
          goto  SQR_END   ; ELSE the process is complete

; Task 3(c): ELSE increment loop count
         incf   COUNT,f

; Task 3(d): Add two to the magic number
         movf   I_L,w
         addlw  2
         btfsc  STATUS,C  ; IF no carry THEN done
          incf  I_H,f     ; ELSE add carry to upper byte I
         movwf  I_L
         goto   SQR_LOOP

; Task 4: Return loop count as the square root
SQR_END  movf   COUNT,w   ; Copy into W
         return
```

The coding in Program 6.11 follows the task list closely. The maximum value of the loop count is FFh, as $\sqrt{65535} = 255$. Thus a single byte at File 35h is reserved for this local variable. Similarly the maximum possible value of the magic number is 511 (1FFh) and so the two registers File 36:7h are reserved for this local variable. This of course means that Task 3(a) entails a double-byte subtraction. The coding is somewhat simplified as the high byte of I, that is I_H, is never more than 01h and so a borrow from the lower byte can be added to a copy of I_H before the high-byte subtract to return the borrow without overflow. If a borrow is generated from this high-byte subtraction the outcome is under zero and the loop is exited. Otherwise COUNT is incremented and I augmented by two. Actually the latter is always twice COUNT plus one, so COUNT is not needed. Instead, on return the 16-bit value I can be shifted once right. This divides by 2 and by throwing away the one that pops out into the carry, effectively subtracts by one – I is always odd and so its least significant bit is always 1. Try coding this alternative arrangement.

Example 6.6

Repeat Example 5.5, which multiplies a byte by ten, but using a software stack for data storage and parameter passing. You may assume that the multiplicand byte is in memory at File 46h.

Solution

The global declarations for the subroutine of Program 6.12 and calling procedure is:

```
PSP     equ   0Ch   ; Holds the Pseudo Stack Pointer
TOS     equ   2Fh   ; File 2Fh is the initial Top Of Stack
INDF    equ   0     ; INDirect File
FSR     equ   04    ; File Select Register
XCAND   equ   46h   ; Multiplicand byte
STATUS  equ   3     ; Status register is File 3
C       equ   0     ; Carry flag is bit0
; The main routine sets up the Pseudo stack pointer (PSP)
MAIN    movlw TOS   ; Set up the PSP
        movwf PSP   ; to the initial Top Of Stack
; ..........................and so on
; Get ready to call up the X10 subroutine
        movf  PSP,w   ; 1st point to current stack position
        movwf FSR
; Now copy multiplicand onto the stack
        movf  XCAND,w ; Copy multiplicand into W
        movwf INDF    ; and onto the stack
        decf  FSR,w   ; Point down one
        call  X10     ; Now call subroutine
; On return PSP is returned to original position
; and product is at PSP+3:PSP+2
NEXT_MAIN  ..... ... ; Continuation of main routine
```

Program 6.12 uses the same technique as the original routine. First the multiplicand is shifted left once to multiply by two and then two further shifts multiplies by eight. The two resulting 16-bit data are then added to give the product. In the same manner as Program 6.6, the File Select Register is moved up and down to point the the appropriate datum as the

Program 6.12 Using a software stack to pass parameters and to provide a workspace. (continued next page).

```
; ************************************************************
; * FUNCTION: Xs byte XCAND by 10 giving double-byte product *
; * EXAMPLE : 64h x 0Ah = 3E8h (100d x 10d = 1000d)          *
; * ENTRY   : Multiplicand pushed into software stack at PSP *
; * EXIT    : Product_H:_L in PSP-3:PSP-2                    *
; ************************************************************
X10     movf   PSP,w     ; Point FSR at current stack position
        movwf  FSR
        clrf   INDF      ; Zero XCAND overflow

; Now multiply by two by shifting 16-bit XCAND left once
        bcf    STATUS,C  ; Clear Carry-in
        incf   FSR,f     ; Point to the XCAND LSB
        rlf    INDF,f    ; Shift left LSB
        decf   FSR,f     ; Point to MSB
        rlf    INDF,f    ; Shift left MSB

; Add to 16-bit subproduct
        incf   FSR,f     ; Point to XCANDx2_L
        movf   INDF,w    ; Get it
        decf   FSR,f     ; Point at PROD_L
        decf   FSR,f
        movwf  INDF      ; Update it with XCANDx2_L
        incf   FSR,f     ; Point to XCANDx2_H
        movf   INDF,w    ; Get it
        decf   FSR,f     ; Point at PROD_H
        decf   FSR,f
        movwf  INDF      ; Update it with XCANDx2_H

; Now shift left twice more to give x8
        incf   FSR,f     ; Point to XCANDx2_L
        incf   FSR,f
        incf   FSR,f
        bcf    STATUS,C  ; Clear Carry-in
        rlf    INDF,f    ; Shift left LSB
        decf   FSR,f     ; Point to MSB
        rlf    INDF,f    ; Shift left MSB
        incf   FSR,f
        rlf    INDF,f    ; Shift left LSB
        decf   FSR,f     ; Point to MSB
        rlf    INDF,f    ; Shift left MSB
```

Program 6.12 (continued.) Using a software stack to pass parameters and to provide a workspace.

```
; Add to 16-bit subproduct
        incf   FSR,f     ; Point to XCANDx8_L
        movf   INDF,w    ; Get it
        decf   FSR,f     ; Point at PROD_L
        decf   FSR,f
        addwf  INDF,f    ; Update it with XCANDx8_L
        incf   FSR,f     ; Point to XCANDx8_H
        btfsc  STATUS,C  ; IF Carry set THEN inc XCANDx8_H
        incf   INDF,f
        movf   INDF,w    ; ELSE get it
        decf   FSR,f     ; Point at PROD_H
        decf   FSR,f
        addwf  INDF,f    ; Update it with XCANDx8_H
        return
; ********************************************************
```

program progresses. The double-byte product can be accessed relative to the Pseudo Stack Pointer by the caller. Unlike Program 6.6 this PSP is not altered when pushing out the multiplicand nor in the subroutine. This is because the subroutine is a dead end in that it can never call another subroutine. Thus a new stack frame need not be formed.

Example 6.7

In order to ensure that the 7-segment decoder subroutine of Program 6.4 does not cause the PCL register to overflow when the offset is added, a programmer has used the directive org (ORiGin – see page 200) to tell the assembler to locate the subroutine at the absolute instruction address $700h$ – as shown in Program 6.13. When the subroutine is tested by calling from another part of the program in the store somewhere lower than $700h$ the system fails and performs unpredictably. What has gone wrong?

Program 6.13 The software 7-segment decoder revisited.

```
        org   700h         ; Start the subroutine at 700h
SVN_SEG addwf PCL,f        ; Add N to PCL giving PC + N
        retlw b'11000000' ; Code for 0
        retlw b'11111001' ; Code for 1
        retlw b'10100100' ; Code for 2
        retlw b'10110000' ; Code for 3
        retlw b'10011001' ; Code for 4
        retlw b'10010010' ; Code for 5
        retlw b'10000010' ; Code for 6
        retlw b'11111000' ; Code for 7
        retlw b'10000000' ; Code for 8
        retlw b'10010000' ; Code for 9
```

Solution

The system goes berserk because on reset the PCLATH register is zeroed. Summoning the subroutine using `call 700h` the value of the PC becomes 0700*h* but the value of the PCLATH register remains unaltered. Later when the `addwf PCL,f` instruction is executed the full 13-bit Program Counter is updated with the bottom eight bits from the PCL register *and* the top five bits from the PCLATH register – as described in Fig. 4.3 on page 86. Thus instead of the execution branching to one of the `retlw` instruction it will jump somewhere in Program memory in the area 0000 – 00FF*h*! This will happen even though adding the offset in W will not cause overflow of the PCL register, as was the original intention.

One way of avoiding this error would be to set the PCLATH to page 7 prior to the call, thus causing the Program Counter to be advanced to location 07NN*h* as desired instead of 00NN*h*.

```
movlw  07h      ; Prepare to point PCL
movwf  PCLATH   ; to page7 of Program store
movf   NN,w     ; Get the decimal number NN into W
call   SVN_SEG  ; Call up the subroutine
```

Even with this kludge the table is limited to 254 entries before the addition overflows the PCL register causing a malfunction. In any case it is considered bad practice for the programmer to specify the absolute location of sections of program code, as it is possible to overwrite code that the assembler places itself. Anyway with large programs it is error prone to try and keep tabs on the location of a myriad of modules. One way around the problems of both large tables and ensuring that the PCLATH register is correctly set, is for the caller to calculate the proper addition of the 13-bit value of the beginning of the subroutine SVN_SEG to the offset and place the top byte of the outcome in the PCLATH register. Of course the PIC is only capable of doing 8-bit arithmetic at a time and so we need to be able to find the values of both the bottom and top bytes of the label SVN_SEG. Fortunately the Microchip assembler has the directives `high` and `low` which can be used to dismember a 13-bit address to facilitate address arithmetic.

```
movlw  high SVN_SEG+1 ; Get hi byte of table start address
movwf  PCLATH         ; Which is the correct Program store page
movlw  low SVN_SEG+1  ; Get the low byte of the table address
addwf  NN,w           ; Add the offset in File NN to it
btfsc  STATUS,C       ; Did this cause a Carry?
 incf  PCLATH,f       ; IF so then means overflow boundary
movf   NN,w           ; Get the offset
call   SVN_SEG
```

In the code segment above the address of the start of the table; that is SVN_SEG+1 is used, as this will be the value of the PC after the opening addwf PC,f instruction. Of course the org 700h instruction used in Program 6.13 can be dispensed with.

This code segment can easily be extended to deal with offsets which are greater than one byte by doing a double-byte addition to update PCL. As before only the lower byte of the offset is sent to the subroutine. In this way look-up tables of any size and located anywhere in Program memory can be implemented subject to the limited size of the Program store.

Self-assessment questions

6.1 Improve the accuracy of the 1-second delay Program 6.9 to within 0.2% by inserting nop instructions into strategic parts of the coding.

6.2 Create a subroutine that will read Port B every hour. You can base it on a 30-second version of Program 6.9. Say why this may not be a good use of the PIC's resources.

6.3 Code a subroutine with the following specification:
- To divide a double-byte dividend by a byte divisor.
- Input DIVIDEND_H:DIVIDEND_L to be passed in File 2E:F*h*.
- Input Divisor to be passed in the Working register.
- Output QUOTIENT_H:QUOTIENT_L to be returned in File 29:A*h*.
- Output Remainder to be returned in W.

Use the subtract until underflow algorithm of Example 3.3 on page 67. Comment on the problem of doing this division in this way.

6.4 Extend Program 6.4 to display A through F. Your solution can use a combination of lower and upper-case glyphs and should be robust.

6.5 Readings of the state of a mechanical switch can be erratic as the contacts will bounce for several milliseconds when closed, thus giving a series of 1s and 0s. Similar considerations apply to electronic devices such as phototransistors when passing through a shadow. Although this problem can be fixed with hardware, it is usually more cost effective to use a software solution.

Devise a subroutine that will return with the stable state of a switch connected to Port B bit 7 as bit 7 of the Working register. Stability is defined as 5000 (1388*h*) reads all giving the same value. The other bits of W are undefined.

6.6 An analog to digital converter is connected to Port B. Repeat SAQ 6.5 but this time defining stability as 1000 identical reads, and returning with the stable digitized analog voltage in W.

6.7 The subroutine in SAQ 6.6 returns the stable value of a noisy digitized signal, assuming 1000 identical values. Using this subroutine, code a main routine that will generate how this stable reading differs from a previous value previously stored in location File 40h. Each bit that differs is to be logic 1. Generate the position of the rightmost change bit in File 41h.

6.8 The subroutine of SAQ 6.6 will not return a value when relatively high-frequency noise is present on the analog signal, as the resulting digital jitter will ensure that 1000 identical readings rarely occur. As an alternative, noise reduction can be obtained by taking the average of multiple readings. If the noise is random then n readings will give a noise improvement of $\sqrt{n}$. Devise a subroutine that will read Port B 256 times and return the 8-bit average in W for an increase in signal to noise ratio of 16.

CHAPTER 7

Interrupt Handling

The subroutines discussed in Chapter 6 are predictable events in that they are called up whenever the program dictates. Real-time situations, defined as where the processor interacts in concert with external physical events, are not as simple as this. Very often something happens beyond the CPU which necessitates precipitate action from the processor. The vast majority of MPU/MCUs have the capability to deal with a range of such events that disrupt their smooth running. In the case of a microcontroller, requests for service may come from an internal peripheral device, such as a timer overflowing, or the completion of an analog to digital conversion, or from a source entirely external to the device in the outside world. At the very least, on reset (a type of external hardware event) the MCU must be able to get (vector) to the first instruction of the main program. In the same manner an external service request or **interrupt** when answered must lead to the start of the special subroutine known as an interrupt service routine.

In this chapter we will be examining how the PIC16F84 handles interrupts originating both internally and externally. The other mid-range PICs handle interrupts in a similar manner, but have a different mix of internal peripheral devices, such as an analog to digital converter, which will be discussed in Part 3 of the text.

After reading this chapter you will:

- Appreciate the need for interrupt handling.
- Appreciate the concept of a vector table as a jumping-off point for reset and interrupt events.
- Follow the sequence of events when the PIC recognizes an interrupt request.
- Understand the principle of latency.
- Have an understanding of the concept of the global interrupt mask.
- Understand the operation of the local interrupt mask and flag pairs and how this is implemented in the INTCON and EECON1 file registers corresponding to the various source of interrupts.
- Be able to write a simple interrupt handler involving the following principles:
  - Context switching.
  - Determination of interrupt source.
  - Return via the `retfie` instruction.

A simple example of a time-sensitive requirement is shown in Fig. 7.1. Here we wish to measure the elapsed time between R points of an electrocardiogram (ECG) signal; by definition an external real-time event. The time resolution is to be 0.1 ms and the maximum peak-peak duration is likely to be no more than 1.5 seconds. In order to measure this time a free-running 16-bit counter clocked at 10 kHz can be used as the time base. As we shall see in Chapter 13, the mid-range PICs have an internal 8-bit counter at File 01 and Fig. 7.1 shows File 3F*h* used as an extension byte to give a total 16-bit count. The details of this configuration are discussed in Program 13.2 on page 370. Here we will assume that the state of the count can be read at any time from these two specified file registers. If the count at the last R point is stored in two spare file registers, then subtraction of the count at the current R point will give the required beat-to-beat duration.

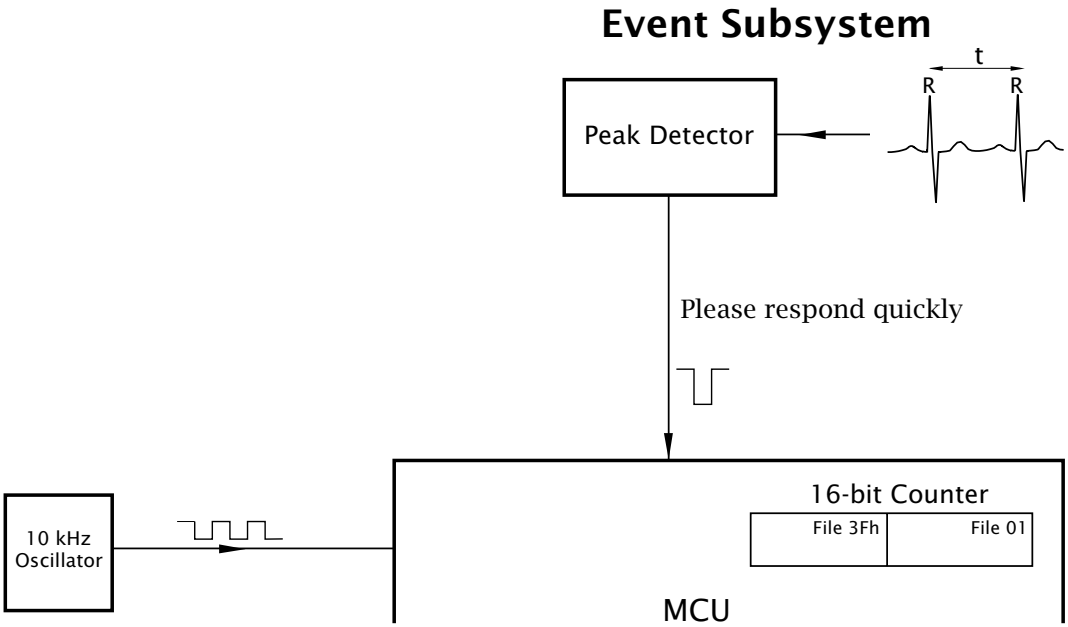


Fig. 7.1 Detecting and measuring an external event.

The next problem is how to detect the signal peak, as by definition the patient's ECG signal is not synchronized to the MCU! One technique is to continually read this signal and perform a peak-detection algorithm to determine the R point. Now this **polling** technique will have to be carried out 10,000 times each second in order to keep to the specified resolution and taking a nominal human heartrate of 60 beats per minute, 99.99% of the time no peak will be detected. Essentially, this means that the processor will spend the vast majority of its processing power just looking out for one event in 10,000.

The alternative approach is to use external hardware who's task is to find the peak signal. That peak-picking hardware could be an analog circuit or even a MCU with an analog to digital converter dedicated to this one task – see Example 14.4 on page 424. Whatever the implementation, the peak-picker sends a signal to the main processor when a R point has been detected. This signal **interrupts** the MCU which must drop whatever it is doing and read the counter within 100 μs if a counter tick is not to be missed.

In the situation where external processes happen in their own good time and are in no way synchronized to the processor, there has to be some way for certain events to **interrupt** the process and direct it to attend to their immediate need. **Polling** a series of outside events is adequate where nothing much happens quickly outside and/or there are few parameters to monitor and little processing to do. The possibility of missing anything important can be reduced by increasing the polling rate, but there comes a time when the MPU does little else but read peripheral data. This resource burnout is especially a problem when there are many signals to poll in a short period of time.

The downside of interrupt-driven real-time monitoring is additional hardware complexity and the greater intricacy of the hardware–software interface. If you are confused, consider the telephone system. It would be possible to have a telephone network where the subscriber would pick up the phone every, say, 5 minutes and ask "Is there anyone there?". Apart from the bother (processing overhead) of doing this,[1] the caller may have got bored and hung up. You could reduce the chance of this happening by increasing the polling rate to, say, once per minute. But you could then end up spending all your time on the phone and, depending on how popular you are, getting only a few hits a day. That is, 99% of your effort is wasted.

This is obviously ridiculous, and in practice an interrupt-driven technique is used so that you only respond when the bell/buzzer sounds. Highly efficient, but at the cost of a lot more complexity for the phone company, as the signalling side of the system can be more demanding than the speech side. There is another problem too, in that you (cf. the processor) have no idea when the phone will ring. And it surely will be at the most inconvenient time. Thus you have to (unless you have an iron will) break off what you are doing at the drop of a hat. For example, if you happen to be in the middle of solving a problem in your head you should save your partial results before responding, so, when finished, you can return to where you left off.

Keeping this apparent randomness (at least as seen by the MCU) in mind the following phases can usually be identified, although the minu-

[1] It would of course make it easier just to ignore the phone…!

tiae of the response to an interrupt request varies considerably from processor to processor.

1. Finishing the current instruction.
2. Automatically saving, at the very least, the state of the Program Counter (PC), which is needed to get back. Some processors also automatically save the Status register and other internal registers at this point.
3. Entering the appropriate interrupt service routine.
4. Executing the defined task.
5. Restoring the processor state and returning to the point in the background program where control was first transferred.

Essentially signalling an interrupt causes the PIC to drop whatever it is doing, save its position in the interrupted **background program** and go to a special subroutine known as an **Interrupt Service Routine** (**ISR**). This **foreground program** is just a subroutine entered at the behest of an external happening.

The mid- and upper-range PIC core can be interrupted by a range of events. For example, the PIC16F84 will respond to service requests from four sources.

1. An external signal at pin 6 (see Fig. 4.2 on page 85) which is labelled INT in this context, but doubles as the Port B bit 0 RB0 pin. The request may be activated optionally by either a rising edge _/‾ or a falling edge ‾_ at this input.
2. An input change at any of the top four Port B (File 6) pins since the last read of this port.
3. By the Timer counter TMR0 (File 1) overflowing FF → 00h.
4. When an internal Data EEPROM write-to action has been completed.

We will look at how each of these can request an interrupt service later in the chapter. However, the PIC's response to an interrupt is functionally identical from whatever source it comes from; so for the moment for simplicity we will assume that some event (maybe the peak-picker in Fig. 7.1) wants service and has pulsed the INT pin.

The PIC's response to such an event is shown in a simplified manner in Fig. 7.2. Essentially the sequence is:

1. The processor samples the interrupt line once in each instruction cycle. If this line is active a latch is set, otherwise it is cleared. This latch is called the **interrupt flag**. Irrespective of the state of this line, the current instruction is always completed; that is execution does not break part way through the instruction, even in a 2-cycle instruction.
2. If the interrupt flag is not active, the PIC simply continues on into the next instruction cycle and the process is repeated.

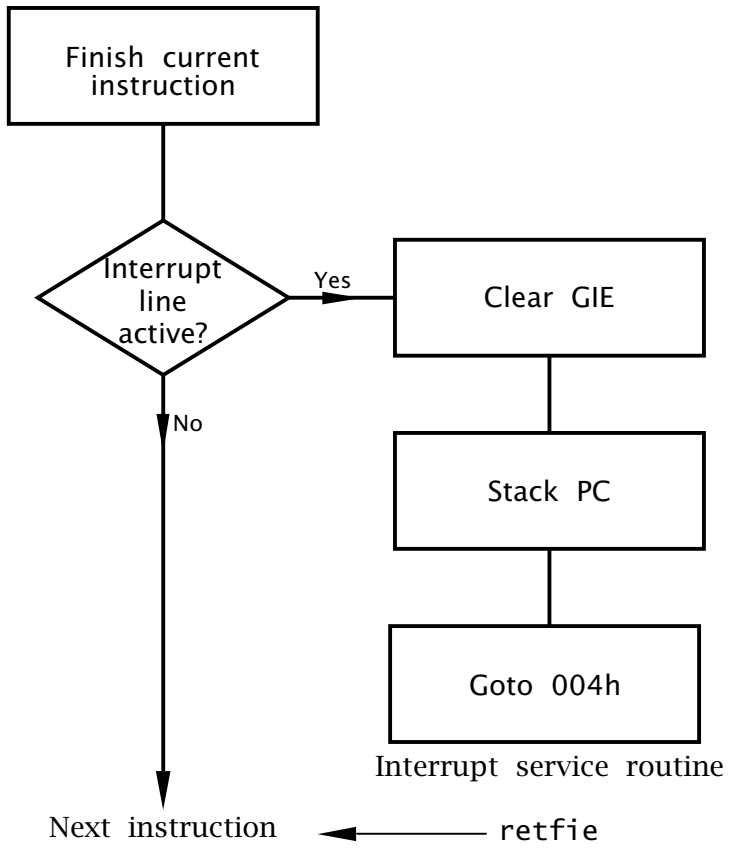


Fig. 7.2 Responding to an interrupt request.

3. If *both* the interrupt flag is set and bit 7 of the INTCON special purpose register (see Fig 7.4) is clear, the next three instruction cycles are involved in moving execution to the interrupt service routine, although the first of these may be the final cycle of a 2-cycle instruction otherwise a dummy cycle, plus two more cycles to flush the pipeline. This 3 to 4-cycle delay from the instant of the hardware INT signal and beginning the execution of the first instruction of the ISR is known as **latency**. It is impossible to be more precise due to the time-random nature of the external request signal which can occur anywhere in the instruction cycle.
4. During this latency period the PIC does three things:
   (a) Bit 7 of the INTerrupt CONtrol register (INTCON at File 0B*h*) is zeroed. This bit is labelled in Fig. 7.4 as General Interrupt Enable (GIE). Once GIE is cleared all further requests for interrupts from *whatever* source are locked out, so an interrupt service process cannot be further interrupted. GIE is an example of an **Interrupt mask** as it is able to mask out interrupt activity. After reset GIE is cleared, so by default interrupt activity is disabled.

(b) The state of the 13-bit Program Counter is pushed into the hardware stack in exactly the same manner as for a `call` instruction – see Fig. 6.3 on page 141. As for subroutines, this is to allow the processor to return to the interrupted background program after the interrupt service routine. As the mid-range PICs have an 8-deep hardware stack, subroutines nested to depth of seven can be called from an ISR.

(c) The first instruction of the ISR is *always* in location `004`h in the Program store. Thus the final step of the sequence is to overwrite the PC with this instruction address, known as the **Interrupt vector**. If the interrupt handling software is elsewhere in Program memory then this entry instruction can of course be a `goto` instruction; see Program 7.1.

5. Like a subroutine, an ISR must be terminated by a Return instruction. However, in this case not only has the PC to be pulled out of the hardware stack to move execution back to the interrupted program but the GIE bit in the INTCON register must be set to re-enable the interrupt capability. This counteracts the resetting of this bit in 4(a) above on entry to the ISR. The Return instruction relevant to this situation is `retfie` (RETurn From Interrupt and Enable). Thus on re-entry to the background program any pending or future interrupts can be serviced.

An ISR differs from a subroutine in more subtle ways than the use of the `retfie` instruction of item 5 above. Some of these differences relate to the logic of the interrupt system and some are due to the pseudo random nature of the interrupts. Discussing the former first, let us examine the logic circuitry relating to the interrupt process.

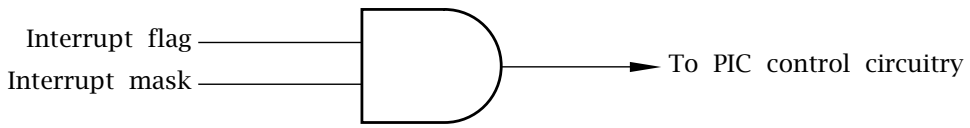


Fig. 7.3 The flag:mask pair.

Each of the four PIC16F84 interrupt sources interact with the processor via two associated control register bits, as shown in Fig. 7.3. The flag bit is set when the related source device requests service. For example, if the Timer 0 overflows FFh → 00h then bit 2 of the INTCON register, labelled T0IF (Timer 0 Interrupt Flag) is set to 1. If the local mask bit is 1 (bit 5 of INTCON labelled T0IE for Timer 0 Interrupt Enable) then the request will go forward to the next layer of interrupt logic. Note that the state of the mask bit does not affect the setting of the associated interrupt flag. Thus if the mask bit is zero, a polling technique can still be

used to determine if an event has occurred by checking the state of the appropriate interrupt flag.

The local mask bit can be written to in the normal way by software. On reset it is zeroed and thus the interrupt from the affiliated source disabled. To set it to one, use the `bsf` instruction. For example, `bsf INTCON,5` to set the T0IE mask. The interrupt flag can also be written to by software, as well as being externally set by the requesting device. The ISR *must* clear it, eg. `bcf INTCON,1`, before return to cancel the request otherwise an endless series of interrupts will occur. This is because on return the interrupt flag will still be set and another interrupt will immediately be set in train.

As there are four sources of interrupt, each flag:mask AND gate must be ORed to give a composite request signal, which when active initiates the CPU's interrupt response. In Fig. 7.4 this ORing process is further gated with the Global mask bit GIE, which is located in bit 7 of INTCON. However, the raw, i.e. pre-globally masked, request signal is used to awaken the processor if it is in a power-down or **sleep** state. As we will see in Chapter 10 the current consumption of the device can be considerably reduced to typically 1 μA if processing is stopped and the PIC is put in a state of suspended animation. For example, monitoring the temperature profile at the bottom of a lake over a period of a year at one hour intervals using a battery-powered data logger requires processing for a tiny proportion of the time. Placing the PIC in this power-down mode after each sample has been taken and stored will reduce the necessary battery capacity. The `sleep` instruction initiates this mode. An interrupt from an outside source, in this case a low-power hourly oscillator, is used to wake the PIC up. As shown, this awakening is independent of the setting of the General mask.

The Timer 0, External and Port B Interrupt flags are located in the INTCON register at bits 2, 1, 0 respectively. These peripherals are common across all mid-range PICs and appear in the same location for these devices. That for the internal Data EEPROM is separately located in the EECON1 Special Purpose Register (SPR) in Bit 4 – see Fig. 15.2 on page 434. Bit EEIF is set whenever a write-data action has been completed.

The Data EEPROM is peculiar to the PIC16F83/4 devices. Other PICs substitute alternative flags for their specialized peripheral devices. For example, the PIC16C71 uses bit 1 of its ADCON0 (A/D CONtrol 0) as the ADIF flag to show that the internal A/D converter has finished its conversion. More sophisticated PICs have more than one interrupting device beyond the three standard ones – INT, Timer 0 and Port B. For instance, the PIC16C74 has eight (plus the standard three) interrupt sources, including two additional timers, a multichannel A/D converter and two serial ports. In this case each such peripheral has its own interrupt flag

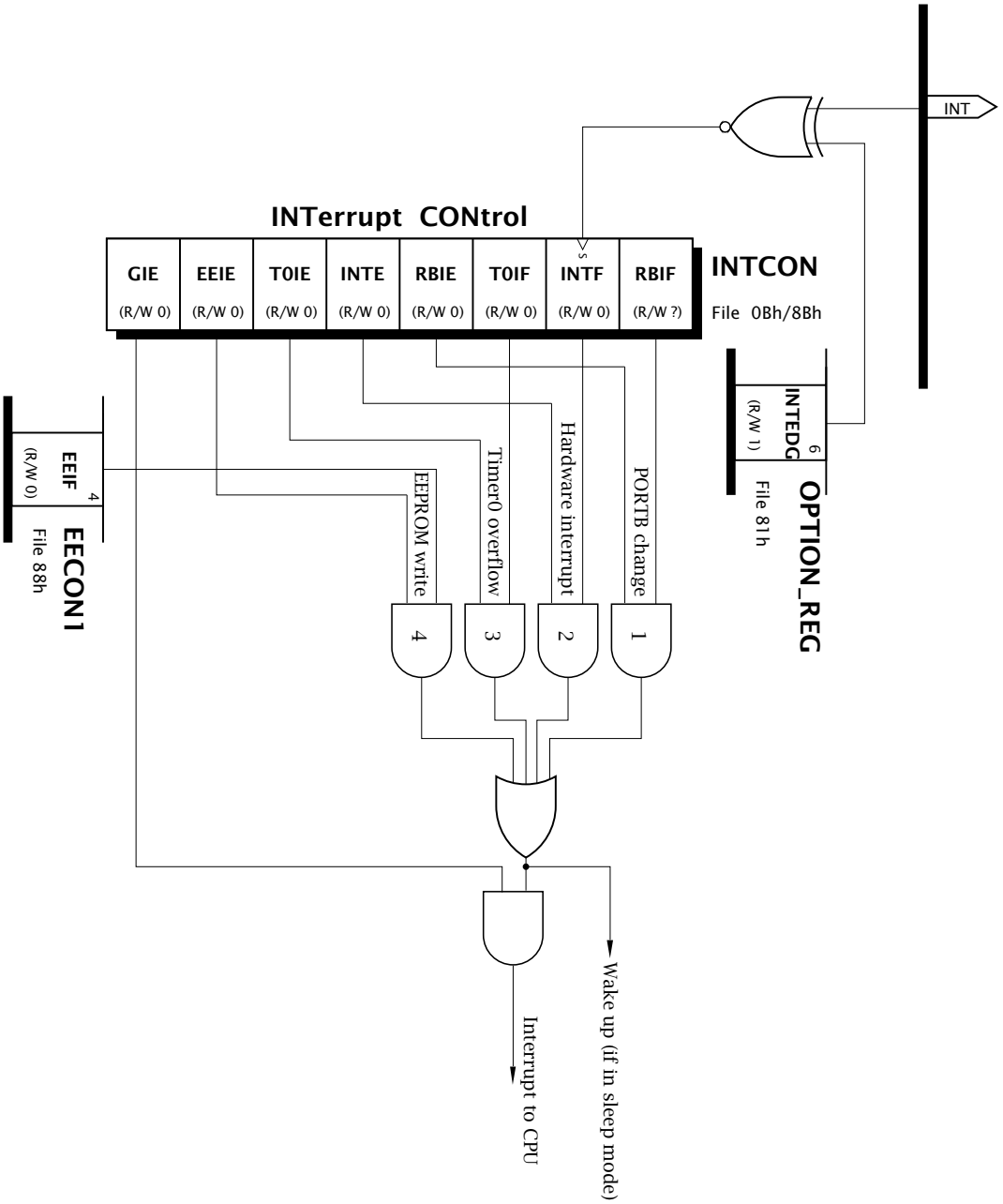


Fig. 7.4 The PIC 16F84's interrupt logic.

together in a Peripheral Interrupt Register, as shown in Fig. 14.10(b) on page 408.

The INTCON register also holds the four local mask bits corresponding to the PIC16F84's standard peripherals, besides the Global mask. The programmer can selectively disable or re-enable one or more interrupting source as desired. Thus if it is undesirable for Timer 0 to interrupt a section of code dealing with, say, multiple-precision arithmetic (see Ex-

ample 7.2) then T0IE can be cleared for the duration of that routine. Other PIC devices replace the EEIE mask by one appropriate to their additional peripheral. Thus the PIC16C71 has ADIE (Analog to Digital Interrupt Enable) as bit 6 of INTCON. PICs with more than one additional peripheral use this bit to enable *all* these extra requests as a single group, called PEIE (PEripheral Interrupt Enable). However, all these extra devices have their own local masks together in a Peripheral Interrupt Enable register, giving three layers of mask – see Fig. 14.10(b) on page 408.

As there is only one common interrupt vector, i.e. at 004*h*, then one of the first tasks the ISR has to do is check which peripheral is calling for help. All interrupt flags can be read, so these can be polled in turn until the one that is set is found. Based on this approach a typical polling sequence could be:

```
STATUS   equ 03    ; The Status register
INTCON   equ 0Bh   ; The INTerrupt CONtrol register
RP0      equ 5     ; bit 5 of which is the RP0 bit
EECON1   equ 88h   ; The EEPROM CONtrol 1 register

         bsf   STATUS,RP0 ; Change to Bank 1 registers
         btfsc INTCON,1   ; Check for external interrupt
          goto EXTERNAL   ; IF set THEN go to INT handler
         btfsc INTCON,2   ; Check for Timer0 interrupt
          goto TIMER0     ; IF set, go to TMR0 handler
         btfsc INTCON,0   ; Check for change at PortB int
          goto CHANGE_B   ; IF set, go to correct handler
         btfsc EECON1,4   ; Check EEPROM write-to inter
          goto EEPROM_WR  ; IF set, go to EEPROM handler

IRQ_EXIT bcf   STATUS,RP0 ; Return to Bank 0 registers
         retfie           ; and return
```

The order of polling gives a priority level if more than one interrupt request should coincide. Thus if both the external hardware and Timer 0 interrupts are active, the former will be processed first. In this case, on return the pending Timer 0 interrupt requests will then be processed – unless another higher-priority interrupt request has occurred. In all instances the appropriate interrupt flag should be cleared, otherwise the interrupt will be generated indefinitely!

Where masks are set, this same polling technique can be used to check on the status of events without using the PIC's interrupt processes. For example, when a byte is written to the Data EEPROM (see Program 15.2 on page 436) the program typically checks the state of EEIF (bit 4 of EECON1) until it is set, then clears it and continues on.

```
W_LOOP  btfss  EECON1,EEIF ; Check state of the EEIF flag
         goto  W_LOOP      ; IF still zero THEN try again

; ELSE continue after clearing the write-to EEPROM flag
        bcf    EECON1,EEIF
```

Figure 7.4 shows additional logic particular to the external INT pin. INTF is set on a falling edge. By interposing an XOR gate as a programmable inverter, as described on page 14, the active edge on the INT pin can be controlled from bit 6 of the Option register – see also Fig. 13.2 page 363. If INTEDG is 0 then INTF will be set on a falling edge at INT whilst a rising edge is active when INTEDGE is 1, which is the default on reset.

Interrupts happen randomly as viewed by the software and thus, unless masked out, may happen at any part of the background software, including in the middle of a subroutine. An ISR foreground routine uses the internal processor registers in the same way as any other software, so conflict over such resources will exist. For example, the background program could just be testing an object when an interrupt occurs. The Skip instruction which follows the test could be dependent on, say, the state of the Zero flag in the Status register. However, the ISR will in all probability alter **Z** and thus on return the background program will execute the skip, oblivious of the fact that execution has been transferred in the interregnum. Any change to **Z** would cause an erroneous branch in the background program. Trying to debug this sort of problem is virtually impossible because the effect of such an interrupt is sporadic as the particular bug depends on the interrupt occurring at just this wrong time and wrong place – something it may do perhaps once a week – and thus is difficult to reproduce.

Another example is illustrated in the polling listing on page 179. Here on entry to the ISR, bit RP0 of the STATUS register (see Fig. 4.6 on page 92) was set to allow access to Bank 1 SPRs. This was necessary as the EEPROM control registers only appear in this bank, whilst both the STATUS and INTCON SPRs are shadowed in both banks. At the exit point, RP0 is cleared to move back to Bank 0. However, this assumes that the background program was in Bank 0 when interrupted. Clearly this is erroneous if an interrupt occurs during an access to Bank 1.

All but the most elementary ISR will need to, at the very least, save the STATUS and Working registers. Generally the programmer sets aside two File registers as temporary storage and for no other use. Traditionally such locations are named with a leading underscore to show that they are used for system purposes and are not to be tampered with by the User's program. In Program 7.1 File 1C*h* and File 1D*h* are labelled `_work` and `_status` to conform to this convention.

Program 7.1 Background program for the pea canning packer.

```
STATUS   equ    03  ; The STATUS register
Z        equ    2   ; and bit2 is the Zero flag
RP0      equ    5   ; and bit5 is the Register Page bit
PORTA    equ    05  ; Port A
TRISA    equ    85h ; whose direction register is in Bank1
INTCON   equ    0Bh ; INTerrupt CONtrol register
INTF     equ    1   ; in which bit1 is the INTerrupt Flag
INTE     equ    4   ; and the associated mask is bit4
GIE      equ    7   ; and the global mask is bit7

_work    equ    1Ch ; Place for the background Working register
_status  equ    1Dh ; and the background STATUS register
EVENT    equ    20h ; Keeps count of cans of peas
; ************************************************************
         org    0           ; Resets here
MAIN     goto BACKGND        ; Go to start of background routine
; ************************************************************
         org    04          ; The interrupt vector
         goto   CAN_COUNT   ; Go to start of foreground ISR
; ************************************************************
; Background program starts by setting up and initialization
BACKGND  bsf    STATUS,RP0 ; Change to Bank1
         bcf    TRISA,0    ; Make bit RA0 an o/p by clearing TRIS0
         bcf    STATUS,RP0 ; Go back to Bank0
         movlw  d'23'      ; Very first value is a dummy dec 23
         movwf  EVENT
         clrf   INTCON     ; Zero any set interrupt flags
         bsf    INTCON,GIE ; Enable all interrupts
         bsf    INTCON,INTE; Enable external INT-pin interrupts

; WHILE event count is less than one DO nothing
LOOP     movf   EVENT,w    ; Get event count
         sublw  1          ; Compare with one
         btfss  STATUS,Z   ; Is it equal?
          goto  LOOP       ; IF not THEN try again ELSE skip

; Now wait until count is back to zero
M_LOOP   movf   EVENT,f    ; Test for zero
         btfss  STATUS,Z   ; Skip IF Zero
          goto  M_LOOP     ; ELSE try again

; Pulse on the 24th can
         bsf    PORTA,0    ; Bring line RA0 high
         call   DELAY      ; Wait for one millisecond
         bcf    POTRA,0    ; and go low again

         goto   LOOP       ; DO forever

; ************************************************************
; * Subroutine delays for a nominal millisecond at 4MHz      *
; ************************************************************
DELAY    movlw  0FFh       ; Count down from 255d: 1~
D_LOOP   addlw  -1         ; Decrement           : (255 x 1)~
         btfss  STATUS,Z   ; Until zero          : (255 X 1) + 1~
          goto  D_LOOP     ;                     : (255 X 2)~
         return            ;                     : 2~
```

Taking as a simple example, consider a conveyer belt in a pea-canning factory. As part of the automatic packing system, a photocell generates a single short pulse for each passing can. The packing machine requires a nominal 1 ms high-going pulse _/‾_ after each batch of 24 cans passes the photocell. Using a PIC16F84 with the photocell sensor connected to the INT pin and the Port A's pin 0 (RA0) driving the packing machine, design both the background and foreground software. Assume that the PIC is being clocked using a 4 MHz crystal.

A suitable background routine is shown in Program 7.1. There are three distinct phases in the code.

Initialization

This phase begins just *after* the Interrupt vector at 004*h*. The PIC always resets to the first instruction in the Program store, i.e. 00*h*, the **Reset vector**. The first instruction is simply `goto BACKGND` where BACKGND is the instruction at 005*h*. Notice how the `org` directive is used to place this label at 005*h*.

As interrupts are automatically disabled on reset, the various File registers and ports are normally set to their initial value at the beginning of the background program before interrupts are enabled. This eliminates the possibility of servicing an interrupt before the initialization code has been completed. The initialization schedule is:

1. All parallel port lines are configured as inputs on reset. To change Port A bit 0 to an output, the associated bit in the TRISA SPR must be cleared. As TRISA is located in Bank 1, RP0 in STATUS is used to switch banks – see page 93. More details are given in Chapter 11.
2. The file register used by the ISR to hold the photocell pulse count is set to an initial value of 23. When the first can passes the sensor, the ISR will set this back to zero thinking that 24 cans have now passed.
3. Clearing all bits in the INTCON register clears all interrupt flags that may have been set since reset. This is important as such flags may be set irrespective of the state of the associated mask bits. Setting the Global Interrupt Enable mask bit now enables the interrupt system and specifically setting INTE enables interrupts from the INT pin.

Main routine

The core of the background software repetitively checks the state of the EVENT file register, which is incremented behind the scenes in the ISR. Initially it waits until EVENT passes a count of one. It then checks for a subsequent zero, which occurs when the ISR detects the first can following event 23. That is the count rolling over 22 → 23 → 0. When a zero is detected RA0 is set high, a 1 ms delay subroutine is called, and then RA0 is brought low again.

The sequence is then repeated indefinitely. The initial wait until EVENT is one ensures a ratchet action, with only one outcome _/‾_ for each zero count in EVENT.

Delay subroutine
The delay subroutine immediately follows the endless loop main routine. The hold off is implemented by decrementing from FFh to zero, with W holding the count. This takes 1026 cycles including the launching call, which at 1 μs per cycle is nominally 1 ms.[2]

Program 7.2 Event counting foreground software.

```
CAN_COUNT
    movwf  _work        ; Save current W reg. in Data memory
    swapf  STATUS,w     ; Get current Status, don't change flags
    movf   _status      ; and put away in Data memory
; ************************************************************
    bcf    INTCON,INTF  ; Clear the hardware interrupt flag
    incf   EVENT,f      ; Record one more event
    movf   EVENT,w      ; Get count
    sublw  d'23'        ; Compare with 23 (23 - EVENT)
    btfss  STATUS,NB    ; IF lower or same THEN skip to finished
     clrf  EVENT        ; ELSE zero can count
; ************************************************************
    swapf  _status,w    ; Untwist & get original Status from mem
    movwf  STATUS
    swapf  _work,f      ; Now get original Working register from
    swapf  _work,w      ; Data memory without altering flags
    retfie              ; and return to interrupted background
```

When an interrupt occurs the PIC always executes the instruction located at 004h, the Interrupt vector. We see from phase 1 of the background program that this causes execution to transfer to the instruction labelled CAN_COUNT, the first instruction in Program 7.2; that is Interrupt $\rightsquigarrow$ 004h $\rightarrow$ CAN_COUNT.

The foreground program, or interrupt service routine, can also be divided into three phases.

Context switching
Both the Working and Status register are saved in the Data store. Firstly, W is copied out to _work. Fortunately movwf does not alter any of the status flags, so the state of STATUS is is still that of the interrupted background program.

Saving this STATUS register is more difficult. The obvious approach is to copy it into W and then out to _status. However, the movf instruction alters the **Z** flag. Instead, we use swapf to copy the datum into W. swapf does not affect the flags but does of course interchange the top and bottom halves of the byte. However, we can untwist them on restoration.

[2] Of course the delay subroutine can be interrupted which will randomly slightly lengthen the delay. In time-critical situations GIE should be zeroed before calling the delay subroutine and set on return.

This process of saving and restoring the state of internal registers (this internal state is called the context) on entry and exit is known as **context switching**. Of course care must be taken that the ISR does not use these locations in Data memory for any other purpose.

Core function
The interrupt flag, INTF in INTCON, is cleared to ensure that on return to the background program another interrupt request is not immediately auctioned…indefinitely. In situations where there is more than one source of interrupts, the various interrupt flags would be polled at this stage of the program – see page 179. There would then be several core routines, each of which would commence by clearing the appropriate interrupt flag.

The functional sector of the core function simply increments the datum EVENT. This is then checked against decimal 23. If it is higher then EVENT is zeroed. The background program looks for this zero as a sign that 24 cans have passed.

Restoring the context
The exit process first restored the original state of the Status register by swapping out of memory into W. This cancels the original `swapf` which was used to save the Status register on entry to the ISR. It is then copied from W into STATUS.

Finally, the Working register has to be restored from Data memory. This process must not alter the newly restored STATUS flag settings. Thus `swapf` is used twice; the first to twist the nybbles out in memory and the second to copy the datum into W and to untwist in the process. The final exit instruction `retfie` does not alter the flag state.

Although our example showed a context change involving STATUS and W, other SPRs can be saved in the same way. For example, if the File Select Register is used by both background and foreground routines then it should be saved and retrieved at the beginning and end of the ISR – see Example 7.4. In general, if the ISR alters any SPR then its original state should be restored on exit. In all cases W needs to be saved first, as it is used as an intermediary for the other transfers, and similarly restored last.

In common with all interrupt-driven systems, special care has to be taken where multiple sources of interrupt are being handled. As an example, consider a certain system receiving interrupt requests from the Timer at, say, 1000 times per second and externally through the INT pin at an irregular rate. If the INT handler takes, say, 4 ms to execute, then on return three Timer requests will have been lost! Some MCUs have interrupt priority logic so that a higher-priority request (the Timer here) will interrupt a lower-priority process (the INT handler). Here the only solu-

tion is to ensure that the latter never takes more than 1 ms to execute.[3] In situations where interrupts from several sources occur, a worst-case scenario budget of execution times (including latency) and interrupt rates must be made. As some of these parameters are related to external events beyond the control of the processor, this can be a non-trivial exercise.

Another problem which frequently arises is dealing with events where multiple-precision data are monitored and changed by both background and foreground routines. Consider as an example a real-time clock (RTC) which updates four register files holding time in the 4-byte multiple-precision format `HOURS:MINUTES:SECONDS:JIFFY`, where the `JIFFY` byte holds tenths of seconds – see Example 7.4. We assume an external 10 Hz oscillator interrupts the PIC ten times per second, and the ISR updates the time-array.

Consider now that this RTC is part of a central heating controller. At `09:00:00:00` hours the water pump is to be toggled from on to off by the background program. One day this has been done and the time is now `09:59:59:09`. The background program, which spends most of its time just looking at the time, reads the hours as `09`. Getting interested, it is just going to read the minutes when the Jiffy oscillator 'ticks'. The MCU is interrupted and the RTC now is updated to `10:00:00:00`. On return the background program now reads in succession `00`, `00`, `00`. Thinking that it is now `09:00:00:00` it toggles the pump off and thereafter the on and off periods are interchanged indefinitely!

Of course it is bad design practice to use a toggle action; instead the pump should be switched off at 9 am rather than toggled. At least in this latter case the harm would be time limited. In general the interrupt handler should be disabled by clearing the appropriate mask where such multiple-precision data manipulation routines are being executed in the background – see also Example 7.2. Any interrupts occurring during this time will be acknowledged when the mask is set, although events could be missed if the masked-out period is too long.

In conclusion, ISRs are similar to subroutines, but keep in mind the following points:

- The ISR should be terminated by `retfie` instead of `return`.
- Any SPRs that are to be altered should be saved on entry and retrieved on exit.
- Parameters cannot be passed to and from the ISR via the Working register. Instead, global variables (data in known memory locations) should be used as required.
- ISRs should be as short as possible, with minimal functionality. This helps in debugging, and helps ensure that other events are not missed.

[3] Rather inelegantly the latter could poll the Timer's interrupt flag `T0IF` at regular intervals.

- Where multiple-byte data objects are being processed by an ISR consideration should be given to disabling the interrupt system during any background access.

Examples

Example 7.1

In a food processing factory, cans of baked beans on a conveyer belt continually pass through a tunnel oven, as shown top of Fig. 7.5, where the contents are sterilized. Photocell detectors are used to sense cans, both entering and leaving the oven. The output of the sensors are logic 1 when the beam is broken.

You are asked to design an interrupt-driven interface for this system, combining the two signals to activate the PIC's *one* INT input. A buzzer connected to Port B's bit RB0 is to be sounded if the number of tins in the oven exceeds four, indicating that a jam has occurred.

Solution

The hardware aspect of this example presents two problems. The first of these involves distinguishing which cell, IN or OUT, generates a request. In Fig. 7.5 each cell clocks an associated D flip flop when the beam is broken. As the D input is permanently tied to logic 1, the clocked flip flop output goes to logic 1. NORing both of these interrupt flags together generates a falling edge at the INT pin if *any* beam is broken.

Both the IN and OUT external flags can be read at Port A pins RA0 and RA1, and this allows the ISR software to distinguish between the two events (can-in and can-out). The appropriate flag can then be reset by toggling the appropriate flip flop reset using two further port lines RA2 and RA3 for Cancel_in and Cancel_out respectively.

One problems remains: If one event follows another before the ISR software has time to reset the appropriate external flip flop, that second event will be missed, as the OR gate will hold INT low. In this situation no further edge can occur and the interrupt system will be permanently disabled! This can be circumvented in software by polling both external flags before exiting the ISR and taking the appropriate action if both bits are not clear.

The interrupt service routine for this hardware configuration is given in Program 7.3. As described on page 181 the context is saved on entry and restored on exit.

The meat of the code simply resets the internal INTF flag and checks each of the external flip flops in turn. Depending on the state of these flip flops, one of three paths through the code is followed:

Program 7.3 Oven safety.

```
OVEN  movwf  _work       ; Save current W reg. in Data memory
      swapf  STATUS,w    ; Get Status, don't change flags
      movf   _status     ; and put away in Data memory
; ************************************************************
CHECK bcf    INTCON,INTF ; Clear the hardware interrupt flag
      btfsc  PORTA,0     ; Check, IN signal?
       goto  IN          ; IF non zero, a can has just gone in
      btfsc  PORTA,1     ; Check for OUT signal
       goto  OUT         ; IF non zero, a can has just gone out
; ************************************************************
; The exit point
      swapf  _status,w   ; Untwist & get old Status from memory
      movwf  STATUS
      swapf  _work,f     ; Now get original W register from
      swapf  _work,w     ; Data memory without altering flags
      retfie             ; and return to interrupted background
; ************************************************************
; The ISR core
IN    incf   EVENT,f     ; Record a can gone in (count up)
      bcf    PORTA,2     ; Clear external IN flag
      bsf    PORTA,2     ; by pulsing its reset
      goto   ALARM       ; and check for alarm situation

OUT   decf   EVENT,f     ; Record a can gone out (count down)
      bcf    PORTA,3     ; Clear external OUT flag
      bsf    PORTA,3     ; by pulsing its reset

ALARM movf   EVENT,w     ; Get Can_count
      addlw  -5          ; Can_count - 5
      btfsc  STATUS,NB   ; IF no borrow THEN sound the alarm
       goto  BUZ_OFF     ; ELSE OK, turn the buzzer off
      bcf    PORTB,0     ; Turn buzzer alarm on
      goto   CHECK       ; and repeat poll of cells flags
BUZ_OFF
      bsf    PORTB,0     ; Turn buzzer off
      goto   CHECK       ; and repeat poll of cell flags
```

1. If pin RA0 is high then a can has broken the IN beam and one is added to the event counter kept in a General-Purpose Register (GPR) in the File store. The external IN flip flop is reset. If the total is greater than four, the buzzer is turned on by bringing RB0 low, otherwise it is turned off. Repeat.
2. If pin RA1 is high then a can has broken the OUT beam and one is taken away from the event counter. This time the external OUT flip flop is reset. Again the total is checked against the boundary of four and the buzzer set to its appropriate state. Repeat.
3. If neither flip flop is set then the ISR exits after restoring the context.

This sequence is repeated whenever actions 1 or 2 have been completed. This ensures that the situation where both beams are broken simultaneously or within a short time window, will be properly serviced.

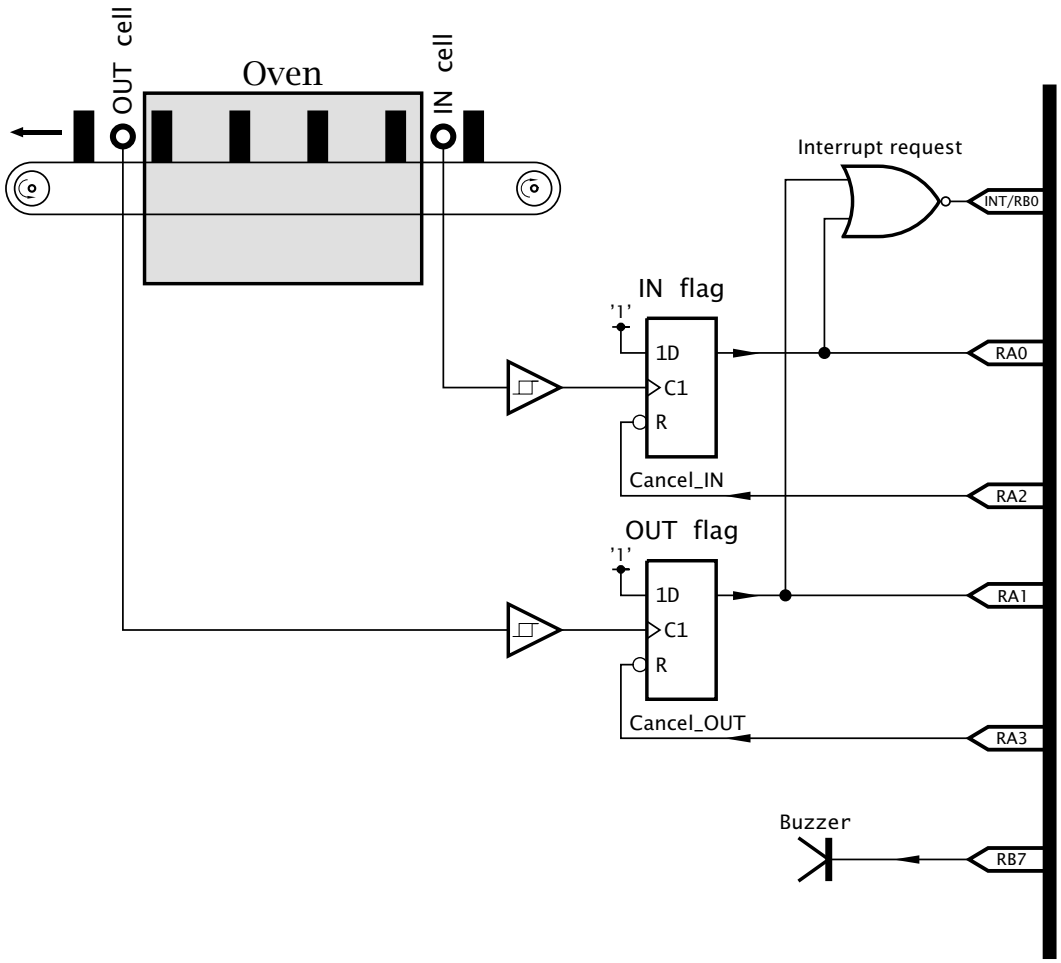


Fig. 7.5 Oven safety hardware.

The main background program is not shown here. It will be similar to that of Program 7.1 in that the various ports will be set up, the event count file register cleared and interrupts enabled. It is likely that this background program will be in charge of sounding the alarm and other consequent tasks rather than implementing this as part of the ISR, in keeping with the philosophy of reducing the size of the foreground code. In a practical system the background program would probably drive a numeric display showing the aggregrate of cans (four was a ridiculous value, chosen for illustrative purposes only) in the oven. Also some means of resetting to a non-zero value after a jam and some sign in the (unlikely) event of a sub-zero count being computed must be facilitated.

Example 7.2

Sensitive routines in the background software, eg. see page 185, may be protected by clearing the GIE (General Interrupt Enable) mask and re-enabling it at the conclusion. However, if an interrupt occurs in the middle of the actual instruction clearing GIE (eg. `bcf INTCON,GIE`) then the interrupt will still be recognized at the instruction's completion and the ISR entered! What effect would this have on the following protected routine and how could this problem be countered?

Solution

On return from the ISR the `retfie` instruction will automatically re-enable the interrupt system by setting GIE, so the supposedly protected routine will be left vulnerable to a further interrupt. Any error will be extremely rare as it depends on the conjunction of the following events:

- An interrupt request occurring coincidentally with the clearing of GIE.
- Another interrupt request happening during execution of the protected code.
- The interrupt service routine causing an error in the protected background process.

Such a rare combination would be difficult to reproduce and thus would be virtually impossible to debug.

To avoid this remote possibility of error, the instruction clearing GIE should be followed with a check that it really is clear. For example:

```
PROTECT  bcf   INTCON,GIE  ; Clear the GIE mask
         btfsc INTCON,GIE  ; Test, is it really clear?
          goto PROTECT     ; IF not THEN do it again
ROUTINE  ..... .....       ; Protected code
         ..... .....
         ..... .....
         bsf   INTCON,GIE  ; Unprotect code
         ..... .....
```

Example 7.3

Interrupt handling anomalies can occur in PICs which do not shadow their general-purpose files registers between banks. We see from Fig. 4.6 on page 92 that all GPRs in the PIC16F84's Bank 0 are shadowed in Bank 1; for instance File $20h$ and File $A0h$ are the same. As an opposite example, the PIC16C74 has GPRs between File $20h$...File $7Fh$ in Bank 0 and between File $A0h$...File FFh in Bank 1; a total of 192 bytes – as shown in Appendix B. These register files are bank specific; for instance File $20h$ is *not* the same as File $A0h$.

The problem arises because the context at the beginning of the ISR will copy W and STATUS into memory in Bank 0 to Bank 3 depending on which bank the processor is in when the background code is interrupted.

If the core of the ISR subsequently switches Bank and then goes back to Bank 0 then an error may occur when restoring the context at the end of the ISR if the bank at this point is not that which the processor was in on entry. For example, if the background program was in Bank 1 at the time of interrupt and then moves to Bank 0, then the restoration at the termination of the ISR will retrieve the incorrect data from Bank 0. Discuss how this problem can be circumvented.

Solution

Microchip suggest two approaches to the context switching problem for cases where the ISR has to change banks and GPRs are bank specific. The first is to keep the background program in Bank 0 at any point of the code where the interrupt is enabled. Access to another bank can of course be made indirectly using the INDF file register – see page 109.

The alternative approach is to save the Working register in whatever bank the interrupt occurs, but the Status register always in Bank 0. When STATUS is retrieved at the end of the ISR, the entry bank state (that is RP1:0 in STATUS – see Fig. 5.1 on page 109) will be restored and the Working (and any other saved) register can then be pulled in from the bank they were actually saved in.

Program 7.4 uses this approach with context change entry and exit routines making no assumptions concerning the entry bank and which enters the core of the ISR code always in Bank 0. Here on entry the state of the Working register is copied into memory in the usual way ignoring the bank in which the processor enters. Although we have defined `_work` in the assembler as File $20h$, if the PIC is in Bank 1 on entry the copy will actually be made at File $A0h$.

The next step is the more complex and depends on the entry bank state.

Bank 0

If the PIC is in Bank 0, as determined by the state of the Status register's RP0 bit, then STATUS is copied into memory at file register `_status` in the normal way using the two instructions at `IN_BANK0`.

Bank 1

If the PIC is in Bank 1 (i.e. RP0 is logic 1) then RP0 is cleared thus moving the processor to bank 0. STATUS is then copied in Data memory at File $21h$, that is `_status`. Before entering the core code, bit 5 of File $21h$ is then set to logic 1, restoring the *copied* version of the original state of RP0 out in memory. This means that on exit when STATUS is restored the PIC will move back into Bank 1.

The core code is always entered with the PIC in Bank 0. At the end of this code the programmer must ensure that the processor is back in Bank 0. In this situation restoring the context is done in the normal way, with STATUS being brought back from `_status` in Bank 0. With this done,

Program 7.4 Saving and restoring the context for the PIC16C74 processor.

```
_work    equ    20h         ; Safekeeping for W @ 20h or A0h
_status  equ    21h         ; Likewise for STATUS

ISR      movwf  _work       ; Save W in Bank0 or Bank1
         btfsc  STATUS,RP0  ; Check which bank PIC is in
          goto  IN_BANK0    ; IF == 0 THEN already in Bank0
; Continue here if PIC is in Bank1 on entry
         bcf    STATUS,RP0  ; Change into Bank0
         swapf  STATUS,w    ; Save STATUS in Bank0
         movwf  _status     ; in the usual way
         bsf    _status,RP0; Set back saved RP0 in memory
         goto   BEGIN       ; and begin the core code

IN_BANK0 swapf  STATUS,w    ; If already in Bank0
         movwf  _status     ; save STATUS in the usual way

; Core code ************************************************
BEGIN    .....   .....      ; Always starts in bank0
         .....   .....
         .....   .....
         .....   .....
; **********************************************************
; Restore context. Processor in Bank0
         swapf  _status,w   ; Untwist & get old Status from memory
         movwf  STATUS      ; which also restores original bank
         swapf  _work,f     ; Now get original W register from
         swapf  _work,w     ; interrupted bank
         retfie             ; and return to interrupted background
```

the PIC is now in its original Bank state and W can be restored in the usual manner from whatever bank it was stored. As this could either be in File 20*h* or File A0*h* then neither file should be used for any other purpose in the ISR to avoid inadvertent corruption.

Example 7.4

On page 185 a central heating real-time clock was discussed. Write a ISR to add one onto the array of file registers holding the four time bytes on each 0.1 s interrupt. Each byte location is to hold its data in a packed binary-coded decimal (BCD) format (see Program 4.1 on page 101) and a 24-hour time representation is to be adopted.

Solution

Each time the PIC enters the ISR one Jiffy must be added to the array of bytes HOURS:MINUTES:SECONDS:JIFFY. The base of each byte count differs in that JIFFY rolls over at a count of ten (i.e. modulo-10), SECONDS

and MINUTES have a modulo-60 count and HOURS is modulo-24. Based on this scenario we have as a task list:

1. Add one onto the JIFFY count.
2. IF this gives 10 THEN zero JIFFY and add one onto the SECONDS count; ELSE goto EXIT.
3. IF this gives 60 THEN zero SECONDS and add one onto the MINUTES count; ELSE goto EXIT.
4. IF this gives 60 THEN zero MINUTES and add one onto the HOURS count; ELSE goto EXIT.
5. IF this gives 24 THEN zero HOURS.
6. EXIT

Coding for this task list is given in Program 7.5. Saving and restoring the context is implemented in the normal way. However, as the File Select Register (FSR) is used in the core of the ISR, it too is saved in a spare file register and retrieved on exit.

Rather than using equ directives to specify the file registers for each of the time array elements and those used to save the STATUS, W and FSR registers I have used the cblock – endc directive (Code BLOCK – END Code block). cblock 20h followed by a list of label names allocates each datum to a successive file register until terminated by endc. Thus _work is allocated to File 20h up to Jiffy at File 26h. Apart from brevity, the main advantage of a single code block over individual equ directives is that when several program sections are concatenated, each with their own code block, the additional variables are simply added to the list. Such additional cblock directives do not specify an explicit start address.

The core of the ISR is sectioned as shown to follow the task list. After each incrementation, the base literal is subtracted from the datum. If they are equal, then the datum is zeroed and the next datum incremented. The alternative of checking the Carry/Not Borrow flag would implement this task if the datum was equal or higher than the base literal, btfss STATUS,C.[4]

The example specified that the datum format should be packed BCD. Thus, 59 minutes should be stored as 0101 1001b or 59h. This means that the incrementation process has to preserve this BCD format. This can be done after a normal increment by checking that the least significant nybble has not gone above nine. If it has, then six is added to correct the situation. As no datum should be above 59 this process is not needed for the upper nybble – Example 4.3 on page 100 gives a task list for a complete packed BCD increment.

As this process needs to be carried out four times (for all except the Jiffy byte which is never greater than nine) then it is best implemented as a subroutine. This is shown in Program 7.6. Here the FSR is pointing to the packed-BCD datum that has to be incremented. This datum is

[4] This is more robust than equality as it is conceivable that due to a software bug a time datum could be set to a value outside the legitimate range.

Program 7.5 Coding the real-time clock ISR.

```
        cblock   20h    ; Reserve space for the following variables
         _work, _status, _fsr, HOURS, MINUTES, SECONDS, JIFFY
        endc
; First save the context ********************************************
RTC     movwf   _work       ; Put away W
        swapf   STATUS,w    ; and the Status register
        movwf   _status
        movf    FSR,w       ; and the File Select Register
        movwf   _fsr
        bcf     INTCON,INTF ; Clear the hardware int flag
; The core code *****************************************************
; Task1
        incf    JIFFY,f     ; Add one onto Jiffy count
; Task2
        movlw   0Ah         ; Compare to ten
        subwf   JIFFY,w
        btfss   STATUS,Z    ; IF equal THEN continue
         goto   EXIT        ; ELSE finished
        clrf    JIFFY       ; ELSE clear Jiffy count
; Task3
        movlw   SECONDS     ; Point FSR to Seconds count
        movwf   FSR
        call    BCD_INC     ; and increment in BCD
        movlw   60h         ; Compare with 0110 0000 (60 BCD)
        subwf   SECONDS,w
        btfss   STATUS,Z    ; IF equal THEN continue
         goto   EXIT        ; ELSE finished
        clrf    SECONDS     ; ELSE clear Seconds count
; Task4
        decf    FSR,f       ; Point FSR to Minutes count
        call    BCD_INC     ; and increment in BCD
        movlw   60h         ; Compare with 0110 0000 (60 BCD)
        subwf   MINUTES,w
        btfss   STATUS,Z    ; IF equal THEN continue
         goto   EXIT        ; ELSE finished
        clrf    MINUTES     ; ELSE clear Minutes count
; Task5
        decf    FSR,f       ; Point FSR to Hours count
        call    BCD_INC     ; and increment in BCD
        movlw   24h         ; Compare with 0010 0100 (24 BCD)
        subwf   HOURS,w
        btfsc   STATUS,Z    ; IF not equal THEN continue
        clrf    HOURS       ; ELSE zero Hours count
; Retrieve the context **********************************************
EXIT    movf    _fsr,w      ; Get the original FSR back
        movwf   FSR
        swapf   _status,w   ; Untwist the original Status reg
        movwf   STATUS
        swapf   _work,f     ; Get the original W reg back
        swapf   _work,w     ; leaving STATUS unchanged
        retfie              ; and return from interrupt
```

Program 7.6 Incrementing a packed-BCD byte with maximum value of 99.

```
; ************************************************************
; * FUNCTION: Adds onto  packed BCD byte, maximum value 99   *
; * ENTRY   : FSR points to byte                             *
; * EXIT    : BCD byte incremented; W and STATUS altered     *
; ************************************************************
BCD_INC  incf   0,f       ; Add one onto pointed-to BCD byte
         movf   0,w       ; Get it down
         addlw  6         ; Add six
         btfss  STATUS,DC ; Check Decimal half Carry
          goto  BCD_EXIT  ; IF none THEN OK to exit
         movwf  0         ; ELSE corrected value put away
BCD_EXIT return
```

simply binary incremented in situ using Indirect addressing. It is then corrected as described. The subroutine assumes that the pointed-to datum is already in a packed-BCD format on entry; it does not convert a natural binary byte to BCD.

Self-assessment questions

7.1 Rewrite Programs 7.1 and 7.2 to deal with a packing quantity of one gross (144). The count is to be kept in packed BCD (Hundreds and Tens:Units) which can be used by the background software to display the can tally.

7.2 What changes to Example 7.1 would you have to make to allow for a maximum value in the oven of 1000?

7.3 Based on Fig. 7.1 design an ISR to perform the following tasks:
- Copy the 16-bit count into two GPRs labelled `TEMP_H` and `TEMP_L`.
- Deduct from the previous count reading located in `LAST_COUNT_H` and `LAST_COUNT_L` and place the difference in `DIFFERENCE_H` and `DIFFERENCE_L`.
- Update the previous count with the new count.
- Set a GPR labelled `NEW` to a non-zero value to signal the background software that a new reading is available. The background routine will clear `NEW` when it has processed the data.

7.4 The speed of a rotating shaft can be measured by using a coded disk to generate a pulse on each angular advance of 10°, which can be used to interrupt a PIC. If the top speed is 20,000 revolutions per minute, what is the absolute maximum duration of the ISR in this

worst-case situation to avoid missing pulses? You may assume a crystal frequency of 4 MHz.

7.5 An electronic tape measure determines distance by pulsing an ultrasonic transmitter and detecting the time it takes for the echo return. The hardware for this echo sounder is shown in Fig. 7.6 and is based on that of Fig. 7.5.

The maximum range is specified as 2.5 meters with a resolution of 1 cm. The speed of sound in air is 344 meters per second at 20°C, which gives a go-return time for one meter of 5.813 ms. Using a 1.72 kHz oscillator as a time base gives one interrupt per 5.813 ms; that is a Jiffy per cm.

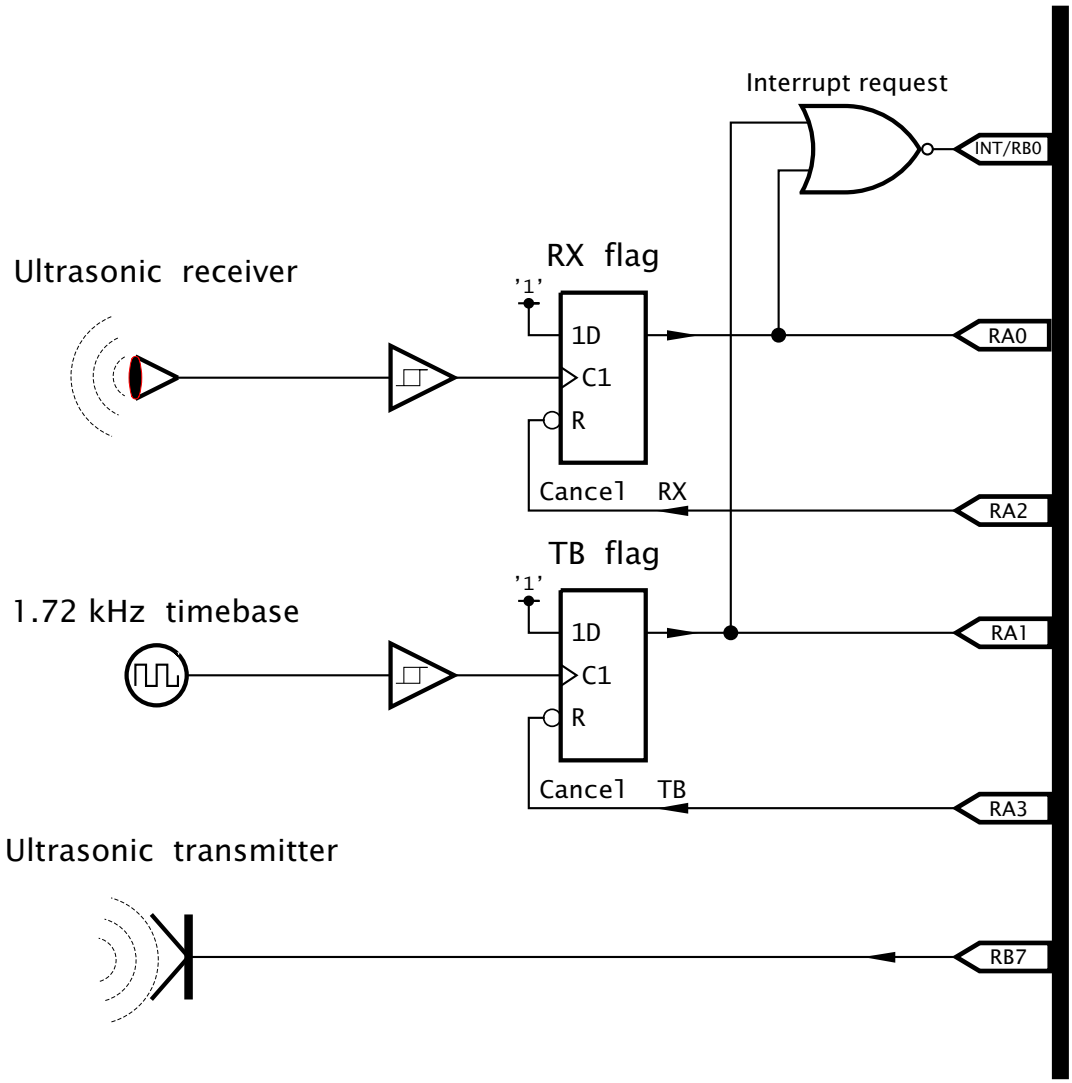


Fig. 7.6 Echo sounding hardware.

Based on this hardware, the software must implement the following task list:

- Background routine
  1. Zero Jiffy count and New flag.
  2. Pulse the sounder.
  3. Wait until New flag is non zero.
  4. Display reading.
  5. Repeat forever.
- Foreground routine.
  1. IF oscillator THEN increment Jiffy count.
  2. IF receiver THEN set New flag to non-zero to tell background program that the Jiffy count is the final value.
  3. Repeat until neither is active.
  4. Return

Code the foreground ISR tasked above using a GPR as a flag labelled NEW to tell the background program that the echo has returned and to read the Jiffy count as the required value. Use Program 7.3 as your model.

7.6 It is proposed to increase the range of the digital echo sounder to 10 meters and resolution to 1 mm. What change in the hardware and software would be required?

7.7 The system in SAQ 7.6 has been built and tested. However, readings seem to shift slowly with time. Oscillator drift is suspected but has been proven to be stable. Thinking laterally, one student wonders if the speed of sound varies with atmospheric conditions. After some research he arrives at the formula for temperature dependence as:

$$V_t = V_0\sqrt{1 + \frac{\Delta t}{273}}$$

where V_0 is the propagation velocity at 20°C and V_t is the velocity at a temperature of t. How much change in temperature Δt will there be to cause an error of 1 mm with the sounder measuring at its maximum range?

CHAPTER 8

Assembly language

We have now been writing programs with gay abandon since Chapter 3. For clarity these listings have been written in a human-readable form. Thus instructions have been represented as a short mnemonic, such as `return` instead of 00000000001000*b*; the file registers similarly have names, such as `INTCON`; lines have been labelled and comments attached. Such symbolic representations are only for human consumption. The MCU knows nothing beyond the binary codes making up operation codes and data, such as shown on page 45.

With the help of the device's instruction set, see Appendix A, it is possible to translate from the human-readable symbolic form to machine-readable binary. This is not particularly difficult for a device such as a PIC that has a reduced set of instructions (RISC) and few address modes. However, it is slow and tedious, especially where programs of a significant length are being coded. Furthermore, it is error prone and difficult to maintain whenever there are changes to be made.

Computers are good at doing boring things quickly and accurately; and translating from symbolic to machine code definitely falls into this category. Here we will briefly look at the various software packages that aid in this translation process.

After reading this chapter you will:

- Know what assembly-level language is and how it relates to machine code.
- Appreciate the advantages of a symbolic representation over machine-readable code.
- Understand the function of the assembler.
- Understand the difference between absolute and relocatable assembly.
- Understand the role of a linker.
- Appreciate the process involved in translating and locating an assembly-level language program to absolute machine code.
- Understand the structure of a machine-code file and the role of the loader program.
- Understand the role of a simulator.

- Appreciate the use of the integrated development environment to automate the interaction of the various software tools needed to convert source code into a programmed MCU device.

The essence of the conversion process is shown in Fig. 8.1. Here the program is prepared by the tame human in symbolic form, digested by the computer and output in machine-readable form. Of course this simple statement belies a rather more complex process, and we want to examine this in just enough detail to help you in writing your programs.

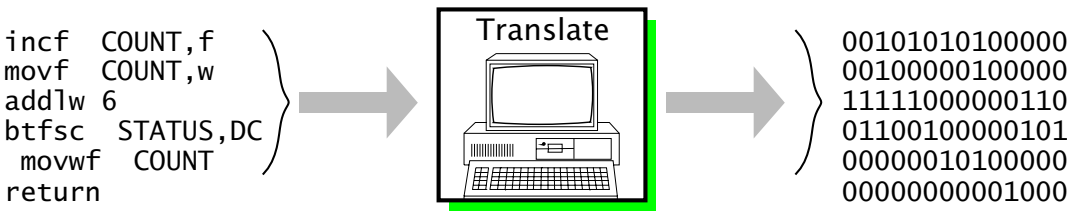


Fig. 8.1 Conversion from assembly-level source code to machine code.

In general the various translator and utility computer packages are written and sold by many software companies, and thus the actual details and procedures differ somewhat between the various commercial products. In the specific case of PIC MCU devices, Microchip Technology Inc. as a matter of policy has always provided their assembly-level software tools free of charge, a large factor in their popularity. For this reason commercial PIC software is relatively rare and what there is usually conforms to the Microchip syntax. For this reason we will illustrate this chapter with the Microchip suite of computer-aided coding tools.

Using the computer to aid in translating code from more user-friendly forms (known as **source code**) to machine-friendly binary code (known as **object code** or **machine code** and loading this into memory began in the late 1940s for mainframe computers. At the very least it permitted the use of higher-order number bases, such as hexadecimal.[1] In this base the code fragment of Fig. 8.1 becomes:

```
0AA0
0820
3E06
1905
00A0
0008
```

A **hexadecimal loader** will translate this into binary and put the code in designated memory locations. This loader might be part of the software

[1] Actually base-8 (octal) was the popular choice for several decades.

in your PIC-EPROM programmer. Hexadecimal coding has little to commend it, except that the number of keystrokes is reduced – but there are more keys – and it is slightly easier to spot certain types of errors.

As a minimum, a symbolic translator, or **assembler**,[2] is required for serious programming. This allows the programmer to use mnemonics for the instructions and internal registers, with names for constants, variables and addresses. The symbolic language used in the source code is known as **assembly language**. Unlike high-level languages, such as **C** or PASCAL, assembly language has a *one-to-one relationship* with the generated machine code, i.e. *one line* of source code produces *one instruction*. As an example, Program 8.1 shows a slightly modified version of Program 6.11 on page 163. This subroutine computes the square root of a 16-bit variable called NUM which has been allocated two bytes in the Data store.

Giving names to addresses and constants is especially valuable for longer programs, which may easily exceed 1000 lines. Together with the use of comments, this makes code easier to debug, develop and maintain. Thus, if we wished to alter the file registers holding the variable NUM from File 20:21h to, say, File 36:37h, then we need only alter the line:

```
cblock 20h
```

to:

```
cblock 36h
```

and then retranslate to machine code. In a program with, say, 50 references to the variable NUM, the alternative of altering all these addresses manually from 20h or 21h (high:low byte) to 36h or 37h respectively is laborious and error prone. In the body of our source code the high byte is referenced as NUM (that is the contents of File 20h) and the lower byte in File 21h as NUM+1, as assemblers can do simple arithmetic on symbolic constants – see page 223.

The pseudo instruction cblock is an example of an **assembler directive**. A directive is a command from the programmer to the assembler concerning its operation or giving a constant a name. We list a small subset of the Microchip assembler directives at the end of the chapter, on page 222; the reader should reference the official manual for a detailed description. Briefly the directives used in Program 8.1 are:

cblock – endc
Rather like a block of equ directives, giving the encapsulated list of label constants starting either at the specified value, eg. 20h, or following on from the last cblock if no address is given. Labelled entities can be deemed to occupy more than one byte by using a colon-delimited size field; for instance NUM:2 for a 2-byte allocation and are normally used to name General-Purpose registers (GPRs).

[2] The name is very old; it refers to the task of translating and *assembling* together the various modules making up a program.

Program 8.1 Absolute assembly-level code for our square-root module.

```
; Global declarations
STATUS     equ   3       ; Status register is File 3
C          equ   0       ; Carry/Not Borrow flag is bit0
           cblock 20h
            NUM:2        ; Number: high byte, low byte
           endc
MAIN       goto   SQR_ROOT
; ************************************************************
; * FUNCTION: Calculates the square root of a 16-bit integer *
; * EXAMPLE : Number = FFFFh (65,535d), Root = FFh (255d)    *
; * ENTRY   : Number in File NUM:NUM+1                       *
; * EXIT    : Root in W. NUM:NUM+1; I:I+1 and COUNT altered  *
; ************************************************************
; Local declarations
         cblock
          I:2, COUNT     ; Magic number hi:lo byte & loop count
         endc
         org    200h     ; Code to begin @ 200h in Program store
SQR_ROOT clrf   COUNT    ; Task 1: Zero loop count
         clrf   I        ; Task 2: Set magic number I to one
         clrf   I+1
         incf   I+1,f
; Task 3: DO
SQR_LOOP movf   I+1,w    ; Task 3(a): Number - I
         subwf  NUM+1,f  ; Subtract lo byte I from lo byte Num
         movf   I,w      ; Get high byte magic number
         btfss  STATUS,C ; Skip if No Borrow out
          addlw 1        ; Return borrow
         subwf  NUM,f    ; Subtract high bytes
; Task 3(b): IF underflow THEN exit
         btfss  STATUS,C ; IF No Borrow THEN continue
          goto  SQR_END  ; ELSE the process is complete
         incf   COUNT,f  ; Task 3(c): ELSE inc loop count
         movf   I+1,w    ; Task 3(d): Add 2 to the magic number
         addlw  2
         btfsc  STATUS,C ; IF no carry THEN done
          incf  I,f      ; ELSE add carry to upper byte I
         movwf  I+1
         goto   SQR_LOOP
SQR_END  movf   COUNT,w  ; Task 4: Return loop count as the root
         return
         end
```

end
Tells the assembler that this is the end of the source code.

equ
Associates a value to a symbol. For instance the assembler replaces the name STATUS by the value 3 anywhere it appears in an instruction operand. Normally used for Special-Purpose registers (SPRs) and bits within file registers.

org
Specifies the start address for following code otherwise the assembler defaults to $000h$ in the Program store. In this program the subroutine SQR_ROOT is originated at $200h$.

Of course symbolic translators demand more computing power than simple hexadecimal loaders, especially in the area of memory and backup store. Prior to the introduction of personal computers in the late 1970s, either mainframe, minicomputers or special-purpose MPU/MCU development systems were required to implement the assembly process. Such implementations were inevitably expensive and inhibited the use of such computer aids, and hand-assembled coding was relatively common.

Translation software thus implements two tasks:

- Conversion of the various instruction mnemonics and labels to their machine-code equivalents.
- The location of the instructions and data in the appropriate memory location.

It is the second of these that is perhaps more difficult to understand.

Program 8.2 is designed to be processed by an **absolute assembler**. Here the programmer uses the directive **org** to tell the assembler to place the code in the specified Program store address. This means that the programmer needs to know where everything is to be placed. This absolute assembly process is shown in Fig. 8.2. Absolute assembly is adequate where a program is contained in a single self-contained file; which is the case for the majority of code in this text. However, real projects often consist of several thousand lines of code and require teamwork. With many modules being written by different people, perhaps also coming in from outside sources and libraries, some means must be found to link the appropriate modules together to give the one executable machine-code file. For example, you may have to call up a division subroutine that Fred has written some time ago. You will not know exactly where in memory this subroutine will reside until the project has been completed. What can you do? Well, a subroutine should have its entry point labelled; say, DIV in this case. You should be able to direct the assembler to give this label the attribute that its absolute value is to be found later by a **linker** program. We will look at this relocatable way of working later on in the chapter.

Most programs running on the low- and mid-range PICs are adequately handled by an absolute assembler. To clarify the process we will take the subroutine of Fig. 8.2 through from the creation of the source file to the final absolute machine-code file.

Editing
Initially the source file must be created using a **text editor**. A text editor differs from a wordprocessor in that no embedded control codes, giving

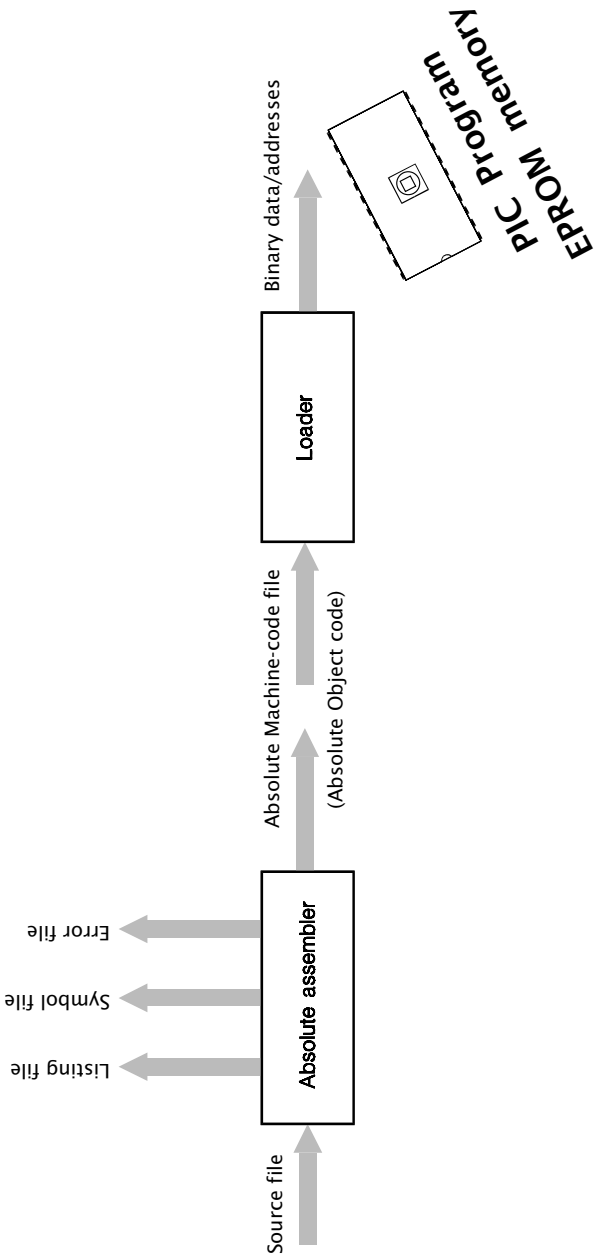
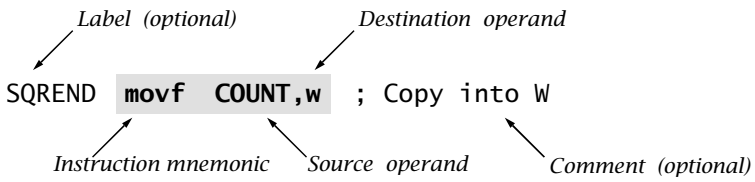


Fig. 8.2 Absolute assembly-level code translation.

formatting and other information, are inserted. For instance, there is no line wrapping; if you want a new line then you hit the [ENT] key. Most operating systems come with a simple text editor; for example, `notepad` for Microsoft's Windows. Third-party products are also available and most wordprocessors have a text mode which can double as a program editor.[3] Microchip-compatible assembly-level source files names have an extension `.src`.

[3]For example, some programs for this book were created using Wordstar 2000 in its non-document format.

The format of a typical line of source code looks like:



With the exception of comment-only lines, all lines must contain an instruction (either executable by the MCU or a directive) and any relevant operand or operands. Any label must begin in column 1, otherwise the first character must be a space or a tab to indicate no label. A label can be up to 32 alphanumeric, underline or question mark characters with the proviso that the first character be an underline or letter. Labels are usually case sensitive. A line label names the Program store address of the first following executable instruction.

An optional comment is delineated by a semicolon, and whole-line comments are permitted – see lines 11–18 of Program 8.1. Comments are ignored by the assembler and are there solely for human-readable documentation. Notes should be copious and should explain what the program is doing, and not simply repeat the instruction. For example:

```
movf I,w    ; Move I into W
```

is a waste of energy:

```
movf I,w    ; Get high byte of magic number
```

is rather more worthwhile. Not, or minimally, commenting source code is a frequent failing, not confined to students. A poorly documented program is difficult to debug and subsequently to alter or extend. The latter is sometimes known as program maintenance.

Space should separate the instruction from any operand. Where there are two operands the source and destination fields are delineated by a comma. In instructions where the destination can be the Working register or the addressed file register, the predefined names `w` or `f` should appear in the destination fields or numbers `0` or `1` respectively. The assembler will default to destination file if omitted.

Assembling

The assembler program will scan the source file checking for syntax errors. If there no such errors the process goes on to translate to absolute object code; which is basically machine code with information concerning the location it is to be placed in Program memory. Syntax errors include such things as referring to labels that don't exist or instructions that are not recognized. The output will include an error file giving any such errors. If there are no syntax errors, a listing file and machine-code file are generated.

In the case of our example the translation was invoked by entering:

```
mpasmwin /aINHX8M /e+ /l+ /c+ /rhex /p16f84 root.asm
```

where `mpasmwin.exe` is the name of the assembler program and `root.asm` is the specified source file. The flags are of the form `/<option>` and may be followed by + or – to enable or disable the option. Thus `/e+` orders the production of an error file, `/l+` likewise for a listing file, `/c+` makes labels case sensitive, `/rhex` specifies the default base radix to be hexadecimal. The flag `/p16f84` tells the assembler to treat the source file as pertaining to the PIC16F84 device. `mpasmwin` can translate code for all PIC devices; whether for 12-, 14- or 16-bit cores.

The **listing file** shown in Table 8.1 reproduces the original source code, with the addition of the hexadecimal location of each instruction and its code. The values of any symbols (such as `NUM` which is listed as File $20h$) is also itemized.

The listing file also provides a symbol table enumerating all symbols/labels defined in the program. The memory usage map gives a graphical representation of Program memory usage. Any warning messages are embedded in the file where they are applicable. For example, if the destination operand `w` or `f` is omitted the assembler will default to the latter and embed a warning message at that instruction in the listing file.

This file has only documentation value and is not executable by the processor.

Executable code

The concluding outcome of any translation process is the **object file**, sometimes known as the **machine-code file**. Once the specified code is in situ in the Program store, it may be run as the executable program.

As can be seen in Table 8.2, such files consist essentially of lines of hexadecimal digits representing the binary machine code, each preceded by the address of the first byte location of the line. This file can be used by the PIC programmer to put the code into Program ROM memory at the correct place. As the location of each code byte is explicitly specified, this type of file is known as **absolute object code**. The software component of the PIC programmer reading, deciphering and placing this code is sometimes called an **absolute loader**.

In the MPU/MCU world there are many different formats in common use. Although most of these de facto standards are manufacturer-specific, in the main they can be used for any brand of MPU/MCU. The format of the machine-code file shown here is known as 8-bit Intel hex and was specified with the flag `/a INHEX8M`.

Let us look at one of the lines in `root.hex` in more detail.

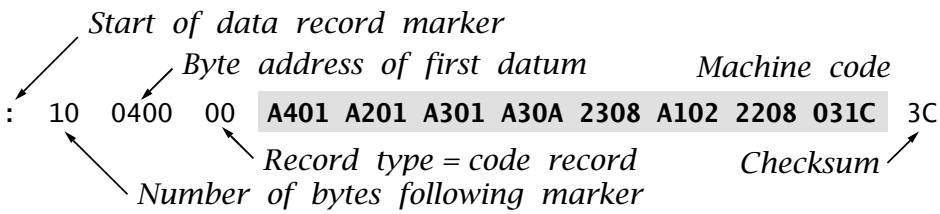


Table 8.1 The listing file `root.lst`. (continued next page).

Listing

```
MPASM 02.20 Released             ROOT.ASM   5-9-1999  14:18:12    PAGE 1
LOC OBJECT CODE LINE SOURCE TEXT
 VALUE
          01 ; Global declarations
 00000003 02 STATUS     equ   3       ; Status register is File 3
 00000000 03 C          equ   0       ; Carry/Not Borrow flag is bit0
          04
          05            cblock 20h
 00000020 06             NUM:2        ; Number: high byte, low byte
          07            endc
          08
0000 2A00 09 MAIN       goto   SQR_ROOT
          10
          11 ; *********************************************************
          12 ; * FUNCTION: Calculates the square root of a 16-bit integer*
          13 ; * EXAMPLE : Number = FFFFh (65,535d), Root = FFh (255d)   *
          14 ; * ENTRY   : Number in File NUM:NUM+1                      *
          15 ; * EXIT    : Root in W. NUM:NUM+1; I:I+1 and COUNT altered *
          16 ; *********************************************************
          17
          18 ; Local declarations
          19            cblock
 00000022 20             I:2, COUNT    ; Magic number hi:lo byte & loop count
          21            endc
          22
0200      23            org    200h
0200 01A4 24 SQR_ROOT clrf   COUNT   ; Task 1: Zero loop count
          25
0201 01A2 26            clrf   I       ; Task 2: Set magic number I to one
0202 01A3 27            clrf   I+1
0203 0AA3 28            incf   I+1,f
          29
          30 ; Task 3: DO
0204 0823 31 SQR_LOOP movf   I+1,w   ; Task 3(a): Number - I
0205 02A1 32            subwf  NUM+1,f ; Subtract lo byte I from lo byte Num
0206 0822 33            movf   I,w     ; Get high byte magic number
0207 1C03 34            btfss  STATUS,C; Skip if No Borrow out
0208 3E01 35             addlw 1       ; Return borrow
0209 02A0 36            subwf  NUM,f   ; Subtract high bytes
          37
          38 ; Task 3(b): IF underflow THEN exit
020A 1C03 39            btfss  STATUS,C; IF No Borrow THEN continue
020B 2A13 40             goto  SQR_END ; ELSE the process is complete
          41
020C 0AA4 42            incf   COUNT,f ; Task 3(c): ELSE inc loop count
          43
020D 0823 44            movf   I+1,w   ; Task 3(d): Add 2 to the magic number
020E 3E02 45            addlw  2
020F 1803 46            btfsc  STATUS,C; IF no carry THEN done
0210 0AA2 47             incf  I,f     ; ELSE add carry to upper byte I
0211 00A3 48            movwf  I+1
0212 2A04 49            goto   SQR_LOOP
          50
0213 0824 51 SQR_END  movf   COUNT,w ; Task 4: Return loop count as root
0214 0008 52            return
          53            end
```

Table 8.1: (continued). The listing file `root.lst`.

```
MPASM 02.20 Released             ROOT.ASM   5-9-1999  14:18:12   PAGE 2

SYMBOL TABLE
  LABEL                             VALUE

C                                 00000000
COUNT                             00000024
I                                 00000022
MAIN                              00000000
NUM                               00000020
SQR_END                           00000213
SQR_LOOP                          00000204
SQR_ROOT                          00000200
STATUS                            00000003
__16F84                           00000001

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

0000 : X--------------- ---------------- ---------------- ----------------
0200 : XXXXXXXXXXXXXXXX XXXXX----------- ---------------- ----------------

All other memory blocks unused.

Program Memory Words Used:    22
Program Memory Words Free:  1002

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

Table 8.2: The absolute 8-bit Intel format object-code file `root.hex`.

```
:02000000002AD4
:10040000A401A201A301A30A2308A1022208031C3C
:10041000013EA002031C132AA40A2308023E03186B
:0A042000A20AA300042A2408080021
:00000001FF
```

The loader recognizes that a record follows when the character `:` is received. The colon is followed by a 2-digit hexadecimal number representing the number of machine-code bytes in the record; $10h = 16d$ in this case. The next four hexadecimal digits represent the starting byte address $0400h$. This is twice the PIC's Program store address of $200h$, as each instruction takes up two bytes. The following 2-digit number is $00h$ for a normal record and $01h$ for the end-of-file record – see the last line of Table 8.2. The core of the record is the machine code with each instruction taking two 2-digit hexadecimal bytes ordered low:high byte. The loader reads this lower byte first (eg. A4) and then 'tacks on' the upper byte (eg. $01h$) giving a 12-, 14- or 16-bit program word as appro-

priate to the target PIC core – eg. 01A4 for a 14-bit core `clrf 24h`.[4] The final byte is known as a **checksum**. The checksum is calculated as the 2's complement of the sum of all preceding bytes in the record; that is −sum. As a check-up on transmission accuracy, the loader adds up all received bytes including this checksum for each record. This received count should give zero if no download error has occurred.

Assemblers are very particular that the syntax is correct. If there are **syntax errors**[5] then an **error file** will be generated. For example, if line 49 was mistakenly entered as:

```
    got  SQRLOOP
```

then the error file of Table 8.3 below is generated.

Table 8.3: The error file

```
Warning[207]  ROOT.ASM  49 : Found label after column 1. (got)
Error[122]    ROOT.ASM  49 : Illegal opcode (SQRLOOP)
```

The assembler does not recognize `got` as an instruction or directive mnemonic and erroneously assumes that it is a label mistakenly not beginning in column 1. On this basis it assumes that `SQRLOOP` is an instruction/directive mnemonic and again does not recognize it.

Most assemblers allow the programmer to define a sequence of processor instructions as a **macro instruction**. Such macro instructions can subsequently be used in a similar manner to native instructions. For example, the following code defines a macro instruction called `Delay_1ms`[6] that implements a 1 ms delay when executed on a PIC running with a 4 MHz crystal. The directive pair **`macro - endm`** is used to enclose the sequence of native instructions which will be substituted when the mnemonic `Delay_1ms` is used anywhere in the subsequent program. The mnemonic will be replaced by the assembler with the defined code. Note that this will be in-line code unlike calling up a subroutine.

[4] Locating the multi-byte code in memory in the Intel way, formatted low:high byte, is known as big-endian (high byte is in the higher memory location) whereas the low-endian arrangement is favored by amongst others, Motorola.

[5] If the assembler announces that there are no errors then there is a tendency to think that the program will work. Unfortunately a lack of syntax errors in no way guarantees that the program will do anything of the sort!

[6] I have capitalized the first letter of all macro instructions to distinguish them from native instructions.

```
Delay_1ms    macro
LOOP         local

             movlw  d'250'   ; Count from 250d
LOOP         addlw  -1       ; Decrement
             btfss  STATUS,Z ; to zero
              goto  LOOP

             endm
```

Where labels are used within the body of the macro, they should be declared using the **local** directive. This means that any conflict with labels where a macro instruction is evoked more than once is avoided.

This example is unusual in that the 'instruction' did not have any operands. Like native instructions, macros can have one or more operands. To see how this is done, consider a macro instruction called Bnz for Branch if Not Zero.[7] Thus the instruction Bnz NEXT causes execution to transfer to the specified label if the Z flag is zero, otherwise continue on as normal. The definition of Bnz is:

```
Bnz    macro   destination

       btfss   STATUS,Z
        goto   destination

       endm
```

Macros can be of any arbitrary complexity and can have any number of comma separated operands. For example, Microchip have available a large number of macros implementing arithmetic operations such as 16×16 and 32×32 multiplication. However, extensive use of macros can make programs difficult to debug, especially when an apparently simple macro instruction hides a number of side effects which alter register contents and flags. A frequent source of error is to precede a macro instruction with a skip instruction, intending to branch around it on some condition. As the macro instruction is in fact a structure of several native instructions, this skip will actually be into the middle of the macro – with dire consequences.

Macro definitions, whether commercial or/and in-house may be collected together as a single file and *included* in the user program using the **include** directive. Thus if your file is called mymacros.mac then the line at the beginning of your program

```
 include   "mymacros.mac"
```

will allow access by the programmer to all macro definitions in the file. The assembler will only generate machine code for any macro instruc-

[7] This is a native instruction for the PIC18CXXX family.

tions actually used by the program. Any macros defined in the included file but not used will have no effect on the final machine code.

One of the major advantages of using an assembler over machine code programming is the use of names for the various file registers and bits therein; for example, INTCON instead of File 0B*h* and GIE in place of 7. In code, such as Program 7.1 on page 181, these equivalences are listed at the head of the source code using equ directives. Although the GPRs will have name labels unique to the particular program, the SPRs and their constituent bits are the same for all programs targeted to a particular type of PIC. Microchip provide files for each PIC, listing all SPRs and bits which can be included in the program's heading; for example

Table 8.4: Part of Microchip's file p16f84.inc.

```
; This header file defines configurations, registers, and other
; useful bits of information for the PIC16F84 microcontroller.
; These names match the data sheets as closely as possible.
;----- Register Files------------------------------------------
INDF                            EQU     H'0000'
TMR0                            EQU     H'0001'
PCL                             EQU     H'0002'
STATUS                          EQU     H'0003'
FSR                             EQU     H'0004'
PORTA                           EQU     H'0005'
PORTB                           EQU     H'0006'
EEDATA                          EQU     H'0008'
EEADR                           EQU     H'0009'
PCLATH                          EQU     H'000A'
INTCON                          EQU     H'000B'
OPTION_REG                      EQU     H'0081'
TRISA                           EQU     H'0085'
TRISB                           EQU     H'0086'
EECON1                          EQU     H'0088'
EECON2                          EQU     H'0089'
;----- STATUS Bits --------------------------------------------
IRP                             EQU     H'0007'
RP1                             EQU     H'0006'
RP0                             EQU     H'0005'
NOT_TO                          EQU     H'0004'
NOT_PD                          EQU     H'0003'
Z                               EQU     H'0002'
DC                              EQU     H'0001'
C                               EQU     H'0000'
;----- INTCON Bits --------------------------------------------
GIE                             EQU     H'0007'
EEIE                            EQU     H'0006'
T0IE                            EQU     H'0005'
INTE                            EQU     H'0004'
RBIE                            EQU     H'0003'
T0IF                            EQU     H'0002'
INTF                            EQU     H'0001'
RBIF                            EQU     H'0000'
```

as used in Program 8.2. For instance, Table 8.4 shows the first part of Microchip's file `p16f84.inc` which we will use for the rest of this text.

The process outlined up to here is known as absolute assembly. Here the source code is in a single file (plus maybe some included files) and the assembler places the resulting machine code in known (i.e. absolute) locations in the Program store. We stated earlier on page 201 that where many modules are involved, often written by different people or/and coming from outside sources and commercial libraries, some means must be found to *link* the appropriate modules together to give the final single absolute executable machine-code file. For example, you may have to call up one of the modules that Fred has written some time ago. You will not know exactly where in memory this module will reside until the project has been completed. What can you do? Well a module should have its entry point labelled; say, `FRED` in this case. Then you should be able to call `FRED` without knowing exactly what address this label represents.

The process used to facilitate this is shown in Fig. 8.3. Central to this modular tie-up is the **linker** program, which satisfies such external cross-references between the modules. Each module's source-code file needs to have been translated into **relocatable object code** prior to the linkage. "Relocatable" means that its final location and various addresses of external labels have yet to be determined. This translation is done by a **relocatable assembler**. Unlike absolute assembly, it is the linker that determines where the machine code is to be located in memory, not the programmer.

Treating the linker as a type of task builder, its main functions are:

- To concatenate code and data from the various input module streams.
- To allocate values to symbolic labels which have not been explicitly given constant values by the programmer – eg. using `equ` directives.
- To generate the absolute machine-code executable file together with any symbol, listing and link-time error files.

In order to allow the linker to do its job, it must have knowledge of the memory architecture of the target processor; basically where the array of general-purpose file registers start and end, where the vectors reside in Program memory and where the code begins and ends. In the case of Microchip's `mplink.exe` linker this information is supplied in the form of a **linker command file**.

A simple example of such a command file for a PIC16F84 is given in Table 8.5. Three directives are used in the file.[8]

codepage

The `codepage` directive is used for program code. These directives are here used to define two regions, one for the Reset and Interrupt vectors

[8] Microchip's MPASM Users Guide with MPLINK and MPLIB manual gives a full list of linker directives.

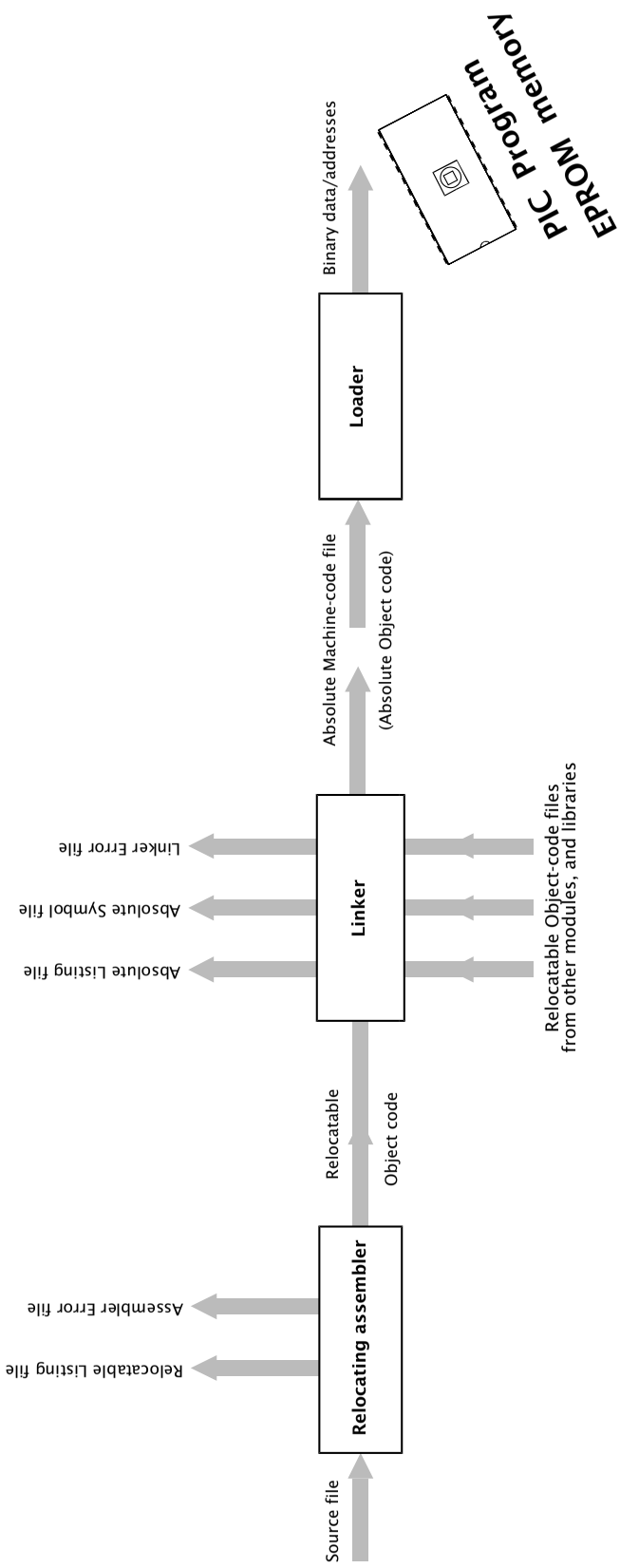


Fig. 8.3 Relocatable assembly-level code translation.

Table 8.5: The pic16f84.lkr linker command file.

```
// File: pic16f84.lkr
// Simple linker command file for 16F84
// Created by S.J. Katzen
// 16/05/1999

CODEPAGE   NAME=vec       START=0x0       END=0x4
CODEPAGE   NAME=program   START=0x5       END=0x3FF

DATABANK   NAME=gprs      START=0x0C      END=0x4F

SECTION    NAME=VECTORS   ROM=vec         // Reset & int vectors
SECTION    NAME=TEXT      ROM=program     // ROM code space
```

in between $000h$ and $004h$ called vec and the other called program to be used for executable code from $005h$ through $3FFh$. Notice the use of the prefix 0x to denote the hexadecimal base. This is the notation used in the **C** language.

databank
This is similar to codepage but is used for variable data in RAM. Here the file register array between File $0Ch$ and File $4Fh$ is called gprs.

section
This linker directive names two code streams. The first called VECTORS will be used by the programmer to store the two vector goto instructions while TEXT is used for the core program code. The source code assembler directive **code** with the appropriate label tells the linker which stream any following code is to be placed; for example, see Program 8.2. As many code sections from any codepage can be created as desired. For instance, all subroutines may be placed together in Program memory by modifying the linker file thus:

```
SECTION  NAME=TEXT        ROM=program  // ROM code space
SECTION  NAME=SUBROUTINES ROM=program  // ROM subroutine stream
```

Sections can be made from DATABANK memory with RAM replacing the ROM attribute. In our case we have not defined a data section and the unlabelled assembler directive **udata** (Uninitialiized DATA) allows space to be reserved for labels in the general-purpose register array; see Program 8.4.

To illustrate the principle of linking we will implement the mathematical function $\sqrt{NUM\_1^2 + NUM\_2^2}$, known as root mean square. There are three teams working on this problem.[9] Tasks have been allocated by the project manager (a fourth person?) as follows:

1. The main function which sequences the steps:

[9] Obviously this is a ridiculously simple problem for team work but it illustrates the principle in a manageable space.

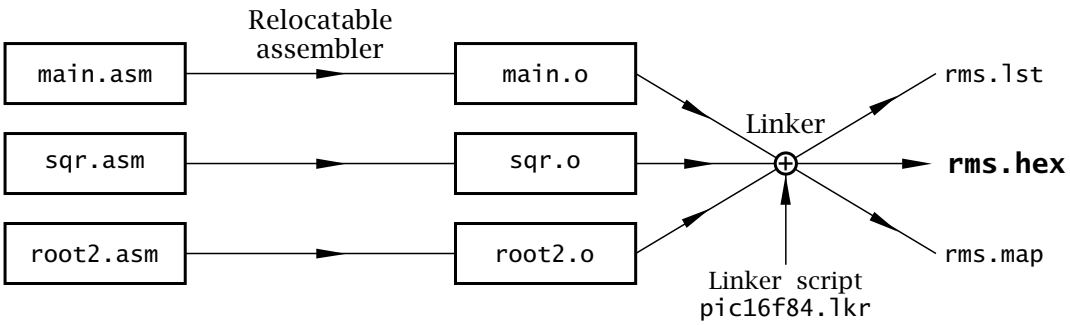


Fig. 8.4 Linking three source files to implement a root mean square program.

   (a) Square NUM_1.
   (b) Square NUM_2.
   (c) Add $\text{NUM\_1}^2 + \text{NUM\_2}^2$
   (d) Square root item (c).
2. Design of a subroutine to square a byte number in the Working register to give a double-byte outcome in two GPRs.
3. Design of a subroutine to evaluate the square root of a double-byte sum and return it in W.

The process based on this decomposition of the task is shown diagrammatically in Fig. 8.4.

The main function is shown in Program 8.2. The program commences with the Reset `goto` instruction and is located in the VECTORS code stream. From the MAIN label onwards, code is located in the TEXT code stream using the directive `TEXT code`. We see from the map file output by the linker in Table 8.6 that MAIN is located at $005h$.

The main routine uses four variables located in general-purpose file. These are placed in uninitialized RAM with the directives `udata` and **`res`**. A single file register is reserved for each of the two input variables NUM_1 and NUM_2 respectively. Two bytes are reserved for SUM which is used to hold the sum NUM_1 + NUM_2. As this is to be the input for the subroutine SQR_ROOT, it is declared `global` at the end of the file. This means that the location is public, that is additional files that are linked together can use the label SUM by declaring it `extern` – i.e. external to the file. Variables not declared thus are 'hidden' from the outside world, i.e. are private (or local) variables. In this manner the directive `extern` at the head of Program 8.2 allows the main routine to call the subroutines SQR_ROOT and SQR without knowing in advance where they are. In the same way the variable SQUARE is used by subroutine SQR to return the square of the byte sent to it in W. Space for this is reserved in a GPR in subroutine SQR and its exact whereabouts is not known by `main.asm` but will be allocated later by the linker. From the map file of Table 8.6 it is finally located in File $11h$.

Program 8.2 The main relocatable source file `main.asm`.

```
            include   "p16f84.inc"
            extern    SQR_ROOT, SQR, SQUARE

            udata                 ; Reserve static data
NUM_1       res       1           ; The first number
NUM_2       res       1           ; The second number
SUM         res       2           ; Two bytes HI:LO for the sum
RMS         res       1           ; One byte for the outcome

VECTORS     code
            goto      MAIN        ; The Reset vector

TEXT        code
MAIN        movf      NUM_1,w     ; Get Number 1
            call      SQR         ; Square it
            movf      SQUARE+1,w  ; Get lower byte
            movwf     SUM+1       ; Is the low byte of sum
            movf      SQUARE,w    ; Get upper byte
            movwf     SUM         ; Is the high byte of sum

            movf      NUM_2,w     ; Now get Number 2
            call      SQR         ; Square it
            movf      SQUARE+1,w  ; Get lower byte
            addwf     SUM+1,f     ; Add to the low byte of sum
            btfsc     STATUS,C    ; Check if produces carry
             incf     SUM,f       ; Add the carry
            movf      SQUARE,w    ; Get upper byte
            addwf     SUM,f       ; Add to the high byte of sum

            call      SQR_ROOT    ; Work out the square root
            movwf     RMS         ; which is the root mean square

            global    SUM
            end
```

The main body of the code follows the task list enumerated above. The value $\texttt{NUM\_1}^2$ is placed in file registers `SUM:SUM+1` to which the computed $\texttt{NUM\_2}^2$ is added. The outcome is then used as input to subroutine `SQR_ROOT` to return the root-mean square byte in `W`. Finally this is copied to the file register named `RMS`, for which a single byte has been reserved in the Data stream.

The subroutine `sqr.asm` of Program 8.3 is based on the subroutine of Program 6.5 on page 152, which multiplies two byte numbers. In this case on entry the contents of the Working register are copied to a file register labelled `X` and a 16-bit version constructed in `X_COPY_H:X_COPY_L`. The shift and add algorithm then evaluates $\texttt{X} \times \texttt{X} = \texttt{X}^2$. These three file registers are allocated with the directive `udata_ovr` (OVeRlay Uninitialized

Program 8.3 The relocatable source file sqr.asm.

```
        include  "p16f84.inc"
; The SQR subroutine
; ***********************************************************
; * FUNCTION: Squares one byte to give a 2-byte result      *
; * EXAMPLE : X = 10h (16), SQUARE = 0100h (256)            *
; * ENTRY   : X in W                                        *
; * EXIT    : SQUARE:2 in shared uninitialized data         *
; ***********************************************************
; Static data
        udata
SQUARE  res     2       ; High:Low byte of square
; Local data
        udata_ovr
X       res     1       ; Place for X
X_COPY_L res    1       ; Holds a copy of X
X_COPY_H res    1       ; Copy X overflow hi byte
TEXT    code
; Task 1: Zero double-byte square
SQR     clrf    SQUARE
        clrf    SQUARE+1
; Task 2: Copy and extend X to 16-bits
        movwf   X           ; Put X away into Data memory
        movwf   X_COPY_L    ; Copy of X
        clrf    X_COPY_H    ; and extend to double byte
; Task 3: DO
  ; Task 3A: Shift X right once
SQR_LOOP bcf    STATUS,C    ; Clear carry
        rrf     X,f         ; Shift
   ; Task 3B: IF Carry == 1 THEN add 16-bit shifted X to square
        btfss   STATUS,C    ; IF C == 1 THEN do addition
         goto   SQR_CONT    ; ELSE skip this task
        movf    X_COPY_L,w  ; DO addition
        addwf   SQUARE+1,f  ; First the low bytes
        btfsc   STATUS,C    ; IF no carry THEN do high bytes
         incf   SQUARE,f    ; ELSE add carry
        movf    X_COPY_H,w  ; Next the high bytes
        addwf   SQUARE,f
     ; Task 3C: Shift 16-bit copy of X right once
SQR_CONT bcf    STATUS,C    ; Zero Carry-in
        rlf     X_COPY_L,f
        rlf     X_COPY_H,f
     ; WHILE X not zero
        movf    X,f         ; Test multiplier for zero
        btfss   STATUS,Z
         goto   SQR_LOOP    ; IF not THEN go again
FINI    return              ; ELSE finished
        global SQUARE, SQR
        end
```

DATA). This is similar to udata but indicates to the linker that file registers allocated in this way can be reused by other modules. In the map file of Table 8.6 we see that X has been allocated File 13*h* as has I, a variable in subroutine SQR_ROOT – see Program 8.3. This makes more efficient use of available Data memory. Variables that are only alive within the subrou-

tine that they are declared in are known in the **C** language as **automatic**, as their space is automatically reallocated as needed. The situation where variable space is preserved is known as **static**. Global variables, such as SQUARE are always static. In this case the variable SQUARE is created by reserving two bytes using the `udata` directive. It is also published using the `global` directive, as is the name of the subroutine.

Program 8.4 The relocatable source file `root2.asm`.

```
          include  "p16f84.inc"
          extern   SUM   ; The 2-byte number Hi:Lo

; Local declarations
          udata_ovr
I         res 2           ; Magic number hi:lo
COUNT     res 1           ; Loop count

TEXT      code

SQR_ROOT clrf   COUNT    ; Task 1: Zero loop count

          clrf   I       ; Task 2: Set magic number I to one
          clrf   I+1
          incf   I+1,f

SQR_LOOP movf   I+1,w    ; Task 3(a): Number - I
          subwf  SUM+1,f  ; Subtract lo byte I from lo byte Num
          movf   I,w      ; Get high byte magic number
          btfss  STATUS,C ; Skip if No Borrow out
           addlw 1        ; Return borrow
          subwf  SUM,f    ; Subtract high bytes

          btfss  STATUS,C ; IF No Borrow THEN continue
           goto  SQR_END  ; ELSE the process is complete

          incf   COUNT,f  ; Task 3(c): ELSE inc loop count

          movf   I+1,w    ; Task 3(d): Add 2 to the magic number
          addlw  2
          btfsc  STATUS,C ; IF no carry THEN done
           incf  I,f      ; ELSE add carry to upper byte I
          movwf  I+1
          goto   SQR_LOOP

SQR_END   movf   COUNT,w  ; Task 4: Return loop count as the root
          return

          global SQR_ROOT
          end
```

The final source file of the trio is the subroutine coded in Program 8.4. This is virtually identical to the absolute equivalent described in Program 8.1. Comparing the two, the `org` directive has been replaced by `TEXT code` and `cblock` by `udata_ovr` for the automatic local data. The data is passed to the subroutine `SQR_ROOT` via the external 2-byte global variable `SUM`, space for which has been allocated in `main.asm`. The subroutine name `SQR_ROOT` is published as global to make it visible to `main.asm`.

Like all source files, `root2.asm` makes use of SPRs such as `STATUS`. For this reason the file `p16f84.inc` of Table 8.4 has been included at the head of the file. As this file comprises a set of `equ` directives, the names thus published are absolute and are not allocated or changed in any way by the linker. Thus the linker map of Table 8.6 does not list such fixed symbols. They are, however, enumerated in the listing file produced by the linker.

In order to link the three source files together, the linker program must be given a command line listing the names of the input object files output by the relocatable assembler, the linker command file and the names of the output map and machine-code file. In the case of our example this was:

```
mplink  p16f84.lkr main.o sqr.o root2.o /m rms.map /o rms.hex
```

which names the output map file `rms.map` and the absolute machine-code file `rms.hex`.

For documentation purposes the linker generates a composite listing file, similar (but more comprehensive) to that of Table 8.1 and an optional map file. The map file of Table 8.6 shows two lists. The first displays information for each section. This includes its name, type, start address, whether the section resides in Program or Data memory and its size in bytes. The Program Memory Usage table shows that 62 bytes of Program memory is used, including the two bytes of the Reset vector `goto` instruction, or around 6% of the possible total.

The second table shows information about the symbols in the composite program. Each symbol's location in either the Program or Data store is given together with the source file where it is defined. Global symbols are noted as `extern`. Local variables are all labelled `static`, including automatic reusable variables such as `COUNT` and `X_COPY_H` both at File 15h.

The final outcome, shown in Table 8.7, is a normal executable machine code file. The format of this file is exactly as described for Table 8.2 and can be loaded into absolute Program memory and run in the normal way.

Table 8.6: The output linker map file `rms.asm`.

```
MPLINK v1.20.00, Linker
Linker Map File - Created Sat Jun  5 16:13:48 1999

                    Section Info
      Section       Type    Address   Location Size(Bytes)
    ---------  ---------  ---------  ---------  ---------
      VECTORS       code     0x0000    program     0x0002
       .cinit    romdata     0x0001    program     0x0004
         TEXT       code     0x0005    program     0x0076
       .udata      udata     0x000c       data     0x0007
   .udata_ovr      udata     0x0013       data     0x0003

                  Program Memory Usage
                    Start         End
                ---------   ---------
                   0x0000      0x0002
                   0x0005      0x003f
62 out of 1024 program words used, memory utilization is 6

                  Symbols - Sorted by Name
        Name    Address   Location    Storage File
    ---------  ---------  ---------  --------- ---------
        FINI     0x002a    program     static SQR.ASM
        MAIN     0x0005    program     static MAIN.ASM
         SQR     0x0015    program     extern SQR.ASM
    SQR_CONT     0x0024    program     static SQR.ASM
     SQR_END     0x003e    program     static ROOT2.ASM
    SQR_LOOP     0x001a    program     static SQR.ASM
    SQR_LOOP     0x002f    program     static ROOT2.ASM
    SQR_ROOT     0x002b    program     extern ROOT2.ASM
       COUNT     0x0015       data     static ROOT2.ASM
           I     0x0013       data     static ROOT2.ASM
       NUM_1     0x000c       data     static MAIN.ASM
       NUM_2     0x000d       data     static MAIN.ASM
         RMS     0x0010       data     static MAIN.ASM
      SQUARE     0x0011       data     extern SQR.ASM
         SUM     0x000e       data     extern MAIN.ASM
           X     0x0013       data     static SQR.ASM
    X_COPY_H     0x0015       data     static SQR.ASM
    X_COPY_L     0x0014       data     static SQR.ASM
```

Developing, testing and debugging software requires a large number of software tools, many of which we have discussed earlier, such as an editor, assembler and linker. In practice there are many other tools such as high-level language compilers (see Chapter 9), simulators and EPROM programmers; shown diagrammatically in Fig. 8.5. Setting up these tools and interacting on an individual basis can be quite complex, especially where products from various manufacturers are involved. In this latter

Table 8.7: The resulting absolute object file `rms.hex`.

```
:020000000528D1
:040002000034003492
:06000A000C08152012088D
:100010008F0011088E000D08152012088F07031895
:100020008E0A11088E072B209000910192019300F7
:1000300094009501031093OC031C242814089207C4
:10004000318910A150891070310940D950D930854
:1000500031D1A2808009501930194019401940A1408BD
:100060008F021308031C013E8E02031C3E28950AD2
:100070001408023E0318930A94002F28150808005C
:00000001FF
```

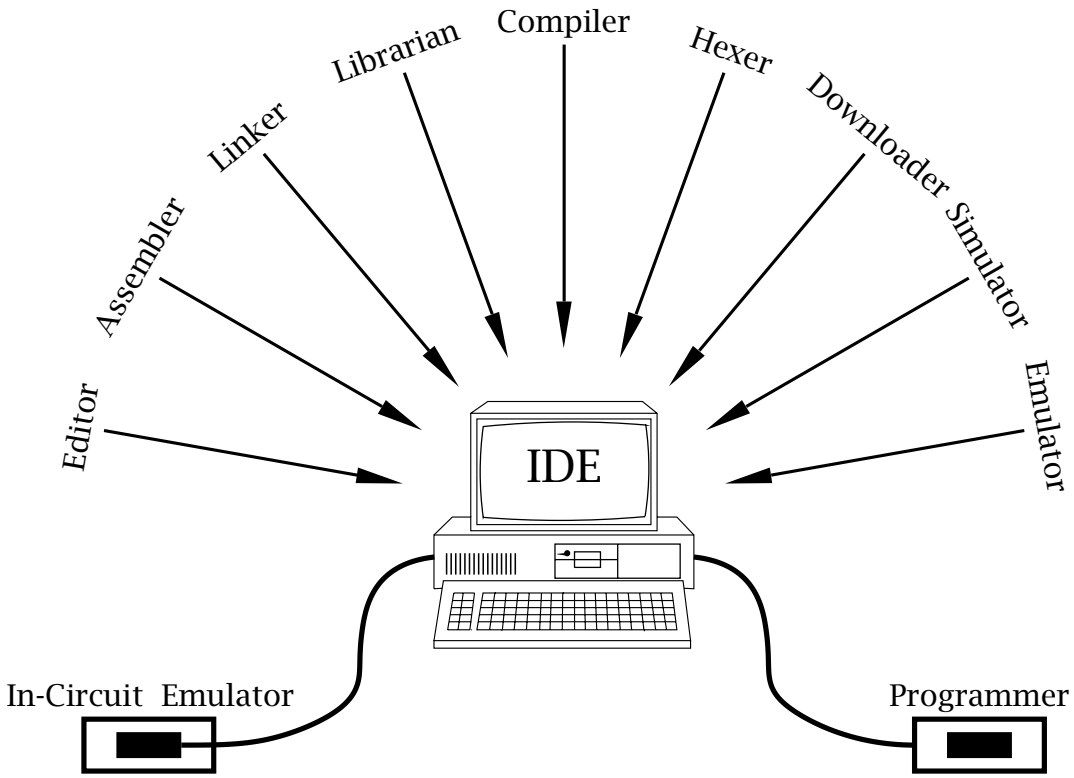


Fig. 8.5 Code building and testing tools.

case, ensuring compatibility between the various intermediate file formats can be a nightmare.

Many software houses designing code development tools provide a graphical environment which integrates and sequences the process in a logical and easy to use manner. Of relevance to the PIC family, Microchip Technology provides a Microsoft Windows-based **Integrated Development Environment** (**IDE**) which brings all compatible code devel-

opment tools under one roof, called MPLAB. Like all Microchip software tools (except **C** compilers) MPLAB is supplied free of charge.

MPLAB integrates Microchip-compatible tools to form a complete software development environment. Among its features are:

- A project manager which groups the specific files related to a project; for example, source, object, simulator, listing and hex files.
- An editor to create source files and linker script files.
- An assembler, linker and librarian to translate source code and create libraries of code, which can be used with the linker without leaving the IDE.
- A simulator to model the instruction execution and I/O on the PC – see Fig 8.7.
- A downloader to work in conjunction with device programmers via the PC's serial port – see Fig. 16.4 on page 472.
- In-circuit emulation software to emulate PIC MCUs in real time in the target hardware. This is accomplished by driving an In-Circuit Emulator (ICE)[10] via the PC's serial or parallel port, replacing the target PIC.

The Microchip manual MPLAB IDE, Simulator, Editor User's Guide gives a MPLAB tutorial and reference details, which are beyond the scope of this book. However, for illustrative purposes two screen shots taken during the development of our previous example linking `main.asm`, `sqr.asm` and `root2.asm` are reproduced in Figs. 8.6 and 8.7.

Figure 8.6 shows the project called `example.pjt` being set up. In the Files window the three source files, which have already been created using the editor, are specified as is the name of the linker script file `pic16f84.lkr` which has also been previously created and saved. The resulting machine-code file is named `rms.hex`.

Once the project is set up in this manner, the sequence of operations, namely:

1. Assemble `main.asm` to give `main.o`.
2. Assemble `sqr.asm` to give `sqr.o`.
3. Assemble `root2.asm` to give `root2.o`.
4. Using `pic16f84.lkr` to link together object files 1, 2 and 3.
5. If no syntax errors, create the absolute executable file of Table 8.7.

can be initiated by choosing from the Project menu (top second left in Fig. 8.7) Make Project. If there are syntax errors an Error window will appear listing errors. Double clicking on any specific error will bring up the relevant Source window with the line in question highlighted.

Once the program has been successfully been linked it may be simulated. Here the PC models the PIC's instruction set and I/O ports and

[10] This is a hardware 'pod' that replaces the PIC chip in the target circuit and allows the PC to take over the running of the system.

Fig. 8.6 MPLAB window showing files selected to assemble, link and simulate Program 8.4.

allows the user to reset the (simulated) PIC, set break points, single step or run continuously. During this process user-selected file registers or the whole of Data memory can be monitored, as can execution time. Of course simulated execution time by the PC will be several orders of magnitude slower than a real PIC.

Figure 8.7 shows the end result of a simulation of our example. In the Watch_1 window are shown the initial values for `NUM_1` and `NUM_2` of $05h$ and $08h$. Values of variables in this window can be set up by the programmer by double-clicking on the variable address. The outcome $\sqrt{5^2 + 8^2} = 9$ (to the nearest integer) is seen in the Watch window as the value of `RMS`. The Watch window is set from the `Window` menu. Just under this window is the Stop-watch window, which shows that the program took 292 cycles to execute with the given data, which for a 4 MHz crystal is 292 μs in real time. After resetting under the `Debug` menu, a breakpoint is set up at the last instruction in `main.asm`. This `movwf RMS` instruction is shown in the screen snapshot greyed out. The program can be 'run' by clicking on the Green Traffic Light icon  in the Simulation tool bar (second icon from the left) or from the `Debug` menu. The Red equivalent

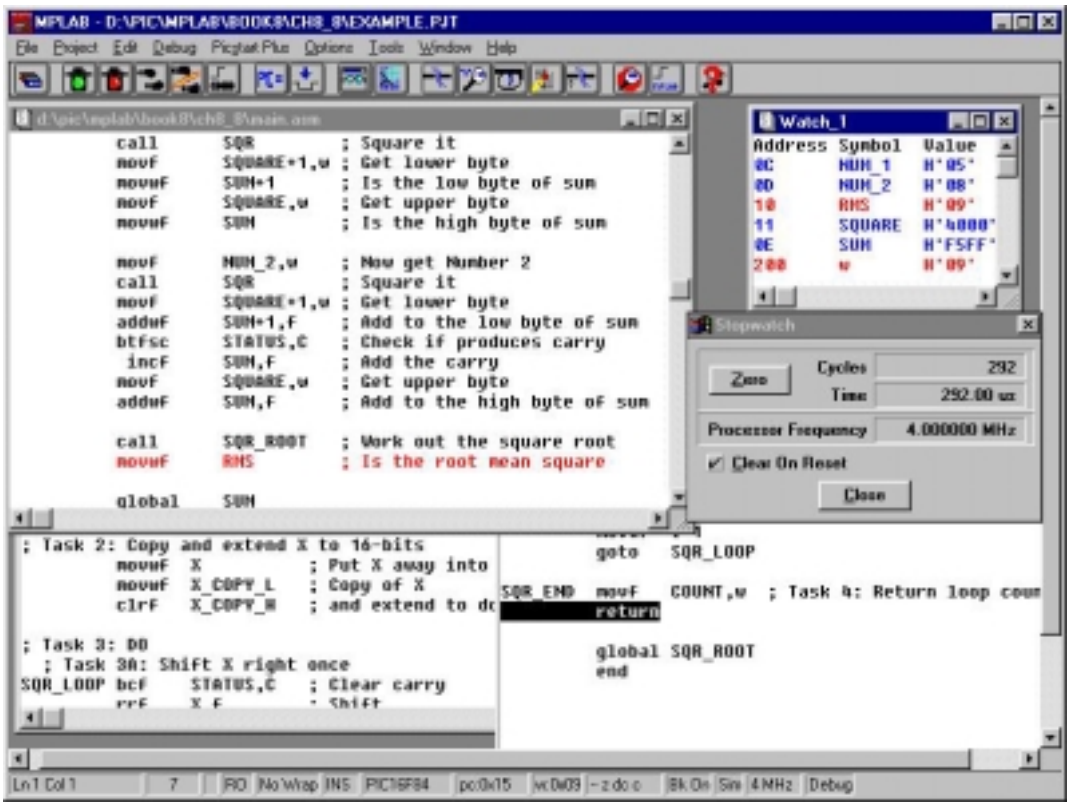


Fig. 8.7 MPLAB screen shot showing the programs selected in Fig. 8.6 being simulated.

 icon next left can be used to pause a run at any time. The  icon is used to single step one instruction at a time.

Simulation will not catch all problems, especially those involving complex hardware/software interaction. However, over 95% of problems are caused by purely software design faults and simulation is good technique for testing and debugging such code.

For example, our code will fail if the total $\texttt{NUM\_1}^2 + \texttt{NUM\_2}^2 > 65,535$, as SUM is only double-byte – see SAQ 8.5. Debugging should always at a first iteration try largest and smallest values of variables. However, correct operation is by no means guaranteed by this test for all possible combinations and sequences of input.

Finally, we review some general information specific to Microchip-compatible assemblers as an aid to reading programs in the rest of the book:

- Number representation.
  - Hexadecimal: Denoted by a following `h`, eg. `41h`, or a leading `h` with the number delineated by quotes, eg. `h'41'` or a `0x` prefix, eg. `0x41`.

The latter is the prefix used in the **C** language to denote this number base.
The assembler normally defaults to this base so some programs show no hexadecimal indicators. However, it is better not to rely on the default behavior.
  - Binary: Denoted by a leading `b` with a quote delimited number; eg. `b'01000001'`.
  - Decimal: Denoted by a leading `d` with a quote delineated number; eg. `d'65'` or a leading period prefix; eg. `.65`.
  - ASCII: Denoted by a quote delimited character; eg. `'A'`.
- Label arithmetic.
  - Current position: `$`; eg. `goto $+2`.
  - Addition: `+`; eg. `goto LOOP+6`.
  - Subtraction: `-`; eg. `goto LOOP-8`.
  - Multiplication: `*`; eg. `subwf LAST*2`.
  - Division: `/`; eg. `subwf LAST/2`.
  - Current position: `$`; eg. `goto $+2`.
- Directives.
  - `org`: Places the following code in Program memory starting from the specified address; eg. `org 0100h`. Defaults to $000h$. Can only be used for absolute assembly.
  - `code`: Counterpart to `org` for relocatable assembly. The actual address of the code stream is defined in the linker's command file. More than one code stream may be defined in the command file and in this case its name appears in the label field; eg. `SUBROUTINES code`.
  - `equ`: Associates a value with a symbol; eg. `PORTB equ 06`. The `#define` directive may be used instead; `#define PORTB 06`.
  - `cblock - endc`: Used in absolute assembly to allocate program variables in Data memory; eg.

```
        cblock  20h
        FRED        ; One byte at 020h for FRED
        JIM:2       ; Two bytes at 021:2h for JIM
        ARRAY:10    ; Ten bytes for ARRAY at 023h - 02Ch
        endc
```

    The address is optional after the first `cblock` use.
  - `udata`: Counterpart to `cblock` for relocatable assembler. The start address for this Data memory stream is in the linker's script file. There may be more than one Data stream defined in this script file in which case its name is published in the label field; eg.

```
SCRATCHPAD  udata          ; Uninitialized data stream
            FRED   res 1   ; Reserve one byte for FRED
            JIM    res 2   ; Reserve two bytes for JIM
            ARRAY  res 10  ; Reserve ten bytes for ARRAY
```

  - `udata _ovr`: OVeRlay Uninitialized DATA is similar to `udata` but the linker tries to reuse File registers for the specified named variables.
  - `res`: Used with `udata` to REServe one or more bytes for a variable in the Data stream.

- `extern`: Publishes the named variables as defined outside the file, to be subsequently resolved by the linker.
- `global`: Publishes the named variables that have been defined (that is space reserved) in the file and that are to be made visible to the linker.
- `macro – endm`: Used to allow the specified enclosed sequence of instructions to be replaced by a new macro instruction; eg.

```
Addf   macro  N,file
       movf   file,w
       addlw  N
       movwf  file
       endm
```

  adds the literal `N` to the specified register file `file`. For example, to add five to File 20*h* the programmer can use the invocation `Addf 5,20h`
- `include`: Used to include the specified file at this point; for example `include "myfile.asm"`.
- `end`: Normally the last line of an assembly-level source file. Tells the assembler to ignore anything following.

Examples

Example 8.1

The following routine effectively exchanges the byte contents of `W` and a file register `F` without needing an additional intermediate file register.

```
    xorwf   F,f    ; [file] <- W^F
    xorwf   F,w    ; W <- W^(W^F) = 0^F = F
    xorwf   F,f    ; [file] <- F^W^F = 0^W = W
```

where ^ denotes eXclusive-OR.

Wrap the given code within a macro to generate a new instruction `Exgwf F` where `F` is the designated file register.

Solution

```
Exgwf   macro   file
        xorwf   file,w
        xorwf   file,f
        xorwf   file,w
        endm
```

Note that this macro instruction will alter the **C** flag according to the outcome of the last instruction.

Example 8.2

The PIC18CXXX family have an instruction `bnc` (Branch if No Carry) which transfers the program execution to the specified destination if the Carry flag is zero. Devise a macro instruction to simulate this for the 12- and 14-bit core families.

Solution

The code fragment below follows the macro on page 208 but with the **C** flag replacing the **Z** flag.

```
Bnc    macro   destination
       btfss   STATUS,C
        goto   destination
       endm
```

Example 8.3

Macros may be nested, that is a macro may use other macros in its definition. For example, consider a macro to create a countdown process that initialises a given GPR and then decrements to zero. Assuming that the macro `Movlf` has already been defined:

```
Movlf macro   literal,destination
      movlw   literal             ; Put the literal in W
      movwf   destination         ; & out to the dest file
      endm
```

write a suitable macro definition.

Solution

One possible solution is:

```
Countdown macro   literal,count_file
          local   C_LOOP               ; A macro label
          Movlf   literal,count_file   ; Initialize counter
C_LOOP    decfsz  count_file,f         ; Decrement
           goto   C_LOOP               ; REPEAT UNTIL zero
          endm
```

The specified file register is firstly initialized to `literal` using the macro `Movlf`. The actual countdown uses the `decfsz` instruction to both decrement the contents of `count_file` and break out of the loop on zero.

Thus the invocation `Countdown d'100',40h` will initialize File 40*h* to decimal 100 and decrement to zero.

Note that both the Working register and STATUS are altered by this macro as well as the target GPR. Side effects are a hazard in using macro instructions, especially if the macro has been designed by someone else and hidden in an include file. At the very least assume that W and STATUS are altered unless known otherwise. Altering banks in a macro is also potentially hazardous.

The macro label is qualified with the `local` directive to ensure that each time a macro is used it does not inject C_LOOP into the assembler's symbol table. If not so qualified then an `Address label duplicated` assembler error will occur.

Example 8.4

The PIC16F84 is unusual in that its GPRs are mirrored across both data banks – see Fig. 4.6 on page 92. More commonly, different banks hold unique GPRs. For example, the PIC16C74 has 96 GPRs in Bank 0 spanning File 20*h*– 7F*h* and another 96 in Bank 1 spanning File 80*h*– FF*h* – see Appendix B. These two banks are not images; thus for instance, File 60*h* in Bank 0 is *not* the same as File E0*h*.

In order to select a file register in Bank 1, the RP0 bit must be set to 1 as described on page 93. For example; to copy W into File E0*h* we have:

```
    bsf   STATUS,RP0    ; Change to Bank 1
    movwf 0E0h          ; Copy W to File E0h
    bcf   STATUS,RP0    ; and move back to Bank 0
```

When using a relocatable assembler, the programmer will not necessarily know which bank the linker has placed a variable. Furthermore, as the suite and mix of component source files changes, the bank may switch back and forth as different phases of the project evolve!

To get around this problem, Microchip-compatible assemblers provide the **`banksel`** directive. This automatically keeps track of the location of the named variable and issues code to make the appropriate change-over. Show how this directive should be used when storing the decimal literals 1, 10, 100 in three GPRs called `var_0`, `var_1` and `var_2` respectively.

Solution

A possible sequence of instructions is shown below. The directive issues either `bsf STATUS,RP0` or `bcf STATUS,RP0` instructions as appropriate before the following instruction. For PICs with four File register banks,[11] both RP1:RP0 bits will be altered by this directive.

[11] 16-bit core PICs can have 16 Data banks of 256-bytes each.

```
    movlw    1     ; The first literal
    banksel  var_0 ; Change to the appropriate bank
    movwf    var_0 ; Do it

    movlw    10    ; Literal ten
    banksel  var_1 ; Change to the appropriate bank
    movwf    var_1 ; Do it

    movlw    100   ; Literal hundred
    banksel  var_2 ; Change to the appropriate bank
    movwf    var_2 ; Do it
```

Where Indirect addressing is used for 4-bank PICs the IRP bit in STATUS must be 0 for Banks 0:1 and 0 for Banks 2:3 – see page 113. This can be implemented using the **bankisel** directive in a similar manner to banksel.

Self-assessment questions

8.1 Design macros to simulate the PIC18XXX family relative conditional Branch instructions bc (Branch on Carry) and bz (Branch if Zero).

8.2 Design a macro of the form Mul XPLIER,XCAND,PRODUCT that will implement the function PRODUCT:2 = VAR1 × VAR2. Hint: Check Program 6.5 on page 152. What do you think are the advantages and disadvantages of using a macro instead of a subroutine in a long implementation like this?

8.3 The goto and call instruction op-codes use an 11-bit address suitable to transfer anywhere within a 2 Kbyte Program store; as illustrated in Fig. 5.4 on page 114. As shown in this diagram, the 13-bit Program Counter is overwritten by the instruction's 11-bit address together with PCLATH[4:3] (PCLATch High byte) File 0A*h* to give a 13-bit destination address.

Some mid-range PICs have a 4 or 8 Kbyte Program store, such as the PIC16C74 and PIC16F876 respectively. These require 12- or 13-bit destination addresses for goto and call instructions making use of the PCLATH[4:3] bits (see page 115) and effectively partitioning up the Program store into corresponding two or four pages. The programmer needs to manipulate these bits before the goto or call instructions to specify the page. For instance, for the PIC16C74 to call a subroutine beginning at FRED which is at address 0B00*h* (i.e. in Page 1) we have:

```
    bsf   PCLATH,3    ; Change to Page1
    call  FRED        ; Go to it
```

In a relocatable program the location of a label, such as FRED is uncertain and in a multi-page PIC may be placed by the linker in any page. To allow the assembler to alter the PCLATH[4:3] bits as appropriate, Microchip compatible assemblers have a directive **pagesel** which must precede any goto or call instruction; rather in the manner of the banksel directive of Example 8.4. Show how you could use this to support a series of calls to subroutines named SUB_0, SUB_1 and SUB_2.

8.4 The banksel approach to selecting a bank is inefficient in that an extra instruction is issued even if the PIC is already in the correct bank. Consider how in a time- or space-critical subroutine this inefficiency can be avoided.

8.5 To be safe, determine a maximum value that NUM_1 and NUM_2 should not exceed to guarantee correct working for our program to calculate the root mean square of the two variables.

8.6 Rewrite the routine main.asm of Program 8.2 and the subroutine root2.asm of Program 8.4 to allow for *all* values of NUM_1 and NUM_2. This will require a 3-byte sum and square root function.

8.7 The following routine based on the macro instruction Movlf of Example 8.3 does not work as intended. COUNT is altered seemingly at random and not consistently with the desired literal 32. Why is this?

```
    movf    COUNT,f     ; Test COUNT for zero
    btfsc   STATUS,Z    ; IF not Zero THEN skip
     Movlf  d'32',COUNT ; ELSE re-initialize it to 32
```

8.8 A programmer with expertise in the Motorola 68HC05 MCU has been converted to the PIC family and wishes to design macros to simulate, amongst others, the following 68HC05 instructions. Note that the Accumulator register in the 68HC05 family is the equivalent to the Working register of the PIC.

lda memory
LoaD Accumulator with data from memory.

lda #data
LoaD Accumulator with literal data.

sta memory
STore Accumulator data into memory.

`tst memory`
TeST memory for zero

`tsta`
TeST Accumulator for zero

Code suitable macros. Why do you think this approach might not be such a good idea?

CHAPTER 9

High-Level Language

All the programs we have written in the last six chapters have been in symbolic assembly language. Whilst assembly-level software is a quantum step up from pure machine-level code (see page 198) nevertheless there is still a one-to-one relationship between machine and assembly-level instructions. This means that the programmer is forced to think in terms of the MCU's internal structure – that is of registers and memory – rather than in terms of the problem algorithm. Although most assemblers have a macro facility, whereby several machine-level instructions can be grouped to form pseudo high-level instructions, this is only tinkering with the difficulty. What is this difficulty with machine-oriented language? In order to improve the effectiveness, quality and reusability of a program, the coding language should be independent of the underlying processor's architecture and should have a syntax more oriented to problem-solving.

We are not going to attempt to teach a high-level language in a single short chapter. However, after completing this chapter you will:

- Understand the need for a high-level language.
- Appreciate the advantages of using a high-level language.
- Understand the problems of using a high-level language for embedded microcontroller applications.
- Be able to write a short program in **C**.

The difficulty in coding large programs in a computer's native language was clearly appreciated within a few years of the introduction of commercial systems. Apart from anything else, computers quickly became obsolete with monotonous regularity, and programs needed to be rewritten for each model introduction. Large applications programs, even at that time, required many thousands of lines of code. Programmers were as rare as hen's teeth and worth their weight in gold. It was quickly deduced that for computers to be a commercial success, a means had to be found to preserve the investment in scarce programmers' time. In developing a universal language, independent of the host hardware, the opportunity would be taken to allow the programmer to express the code in a more natural syntax related to problem-solving rather than in terms of memory, registers and flags.

Of course there are many different classes of problem tasks which have to be coded, so a large number of languages have been developed since.[1] Amongst the first were Fortran (FORmula TRANslation) and COBOL (COmmon Business Oriented Language) in the early 1950s. The former has a syntax that is oriented to scientific problems and the latter to business applications. Despite being around for over 40 years, the inertia of the many millions of lines of code written has made sure that many applications are still written in these antique languages. Other popular languages include Algol (ALGOrithmic Language), BASIC, Pascal, Modula, Ada, **C**, **C++** and Java.

Although writing programs in a high-level language may be easier and more productive for the programmer, the process of translation from the high-level source code to the target machine code is much more complex than the assembly process described in Chapter 8. The translation package for this purpose is called a **compiler** and the process **compilation**.

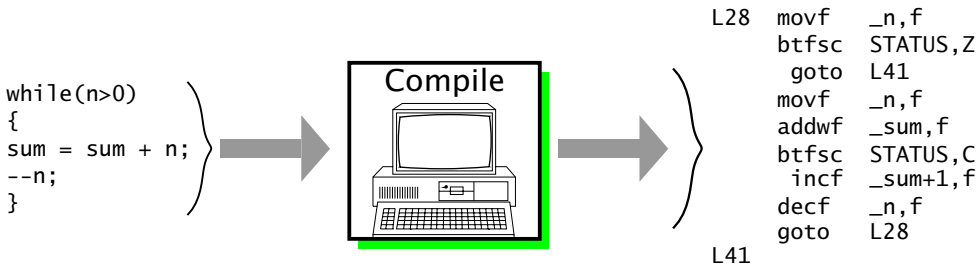
The complexity and cost of a compiler was acceptable on the relatively powerful and extremely expensive mainframe computers of that time. However, until the mid-1980s the use of high-level languages as source code was virtually unknown for MPU-controlled circuitry. In the last decade the easy availability of relatively powerful and cheap personal computers and workstations, capable of running compilers, together with the growing power of MPU/MCU targets and financial importance of this market, is such that the majority of software written for such targets is now in a high-level language.

If you are going to code a task in a high-level language to run in a system with an **embedded** MCU; for example, a washing-machine controller, then the process is roughly as follows.

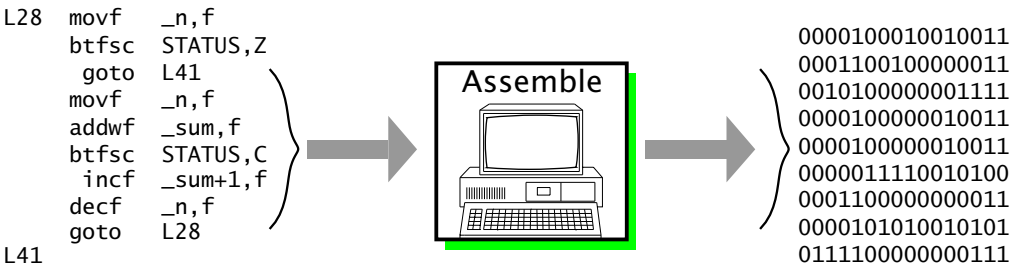
1. Take the problem specification and break it up into a series of modules, each with a well-defined task and set of input and output data.
2. Devise a coding to implement the task for each module.
3. Create a source file using an editor in the appropriate high-level syntax.
4. Compile the source file to its assembly-level equivalent.
5. Assemble and link to the machine-code file.
6. Download the machine code to the target's program memory.
7. Execute, test and debug.

This is virtually identical to the process outlined in Fig. 8.3 on page 211, but with the extra step of compilation. Some compilers go directly from the source file to the machine-code file; however, the extra flexibility of going through the assembly-level phase, as shown in Fig. 9.1, is nearly universal when embedded MPU/MCU circuitry is targeted.

[1] A popular definition of a computer scientist is one who, when presented with a problem to solve, invents a new language instead!



(a) First, compile to assembly-level code.



(b) Second, assemble-link to machine code.

Fig. 9.1 Conversion from high-level source code to machine code.

The choice of a high-level language for embedded targets is crucial. Of major importance is the size of the machine code generated by a high-level language task implementation as compared with the equivalent assembly-level solution. Most embedded MCU circuitry is lean and mean, such as the remote controller for your television. Lean translates to physically small and mean maps to low processing power and memory capacity – and cost! Most low-cost MCUs have a low-capability processor with a few hundred bytes of RAM and a few kilobytes of ROM Program store at best. Thus to be of any use the high-level language and the compiler must generate code, that if not as efficient as assembly-level (low-level), at least is in the same ball park.[2]

By far the most common high-level language used to source code for embedded MPU/MCU circuitry is **C**. Historically **C** was developed as a language for writing operating systems. At its simplest level, an operating system (OS) is a program which makes the detailed hardware operation of the computer's terminals, such as keyboard and disk organization, invisible to the operator. As such, the writer of an OS must be able to poke about the various registers and memory of the computer's peripherals and easily integrate with assembly-level driver routines. As conventional high-level languages and their compilers were profligate with resources, depending on a rich and fast environment, assembly language was mandatory up to the early 1970s, giving intimate machine contact and tight fast code. However, the sheer size of such a project means that

[2] In the author's experience a code size increase factor of $\times 1.25 \cdots \times 2.5$ is typical.

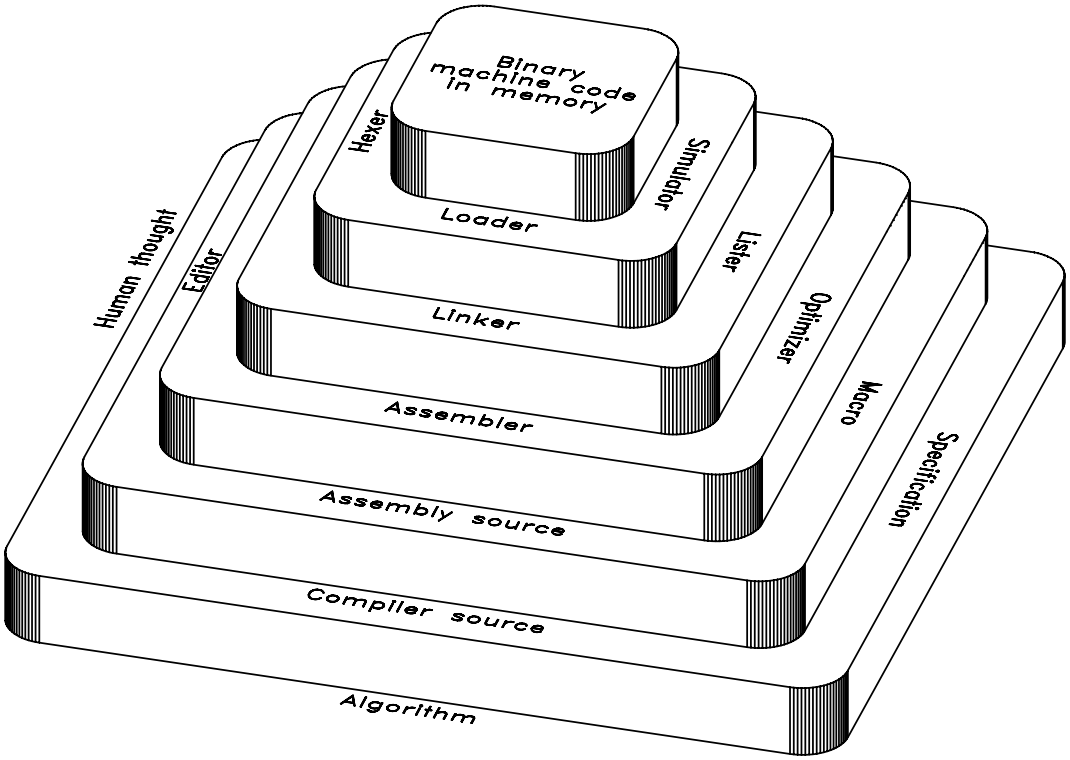


Fig. 9.2 Onion skin view of the steps leading to an executable program.

it is likely to be a team effort, with all the difficulties in integrating the code and foibles of several people. A great deal of self-discipline and skill is demanded of such personnel, as is attention to documentation. Even with all this, the final result cannot be easily transplanted to machines with other processors, needing a nearly complete rewrite.

In the early 1970s, Ken Thompson – an employee at Bell Laboratories – developed the first version of the UNIX operating system. This was written in assembler language for a DEC PDP7 minicomputer. In an attempt to promote the use of this operating system (OS) within the company, some work was done in rewriting UNIX in a high-level language. The language CPL (Combined Programming Language) had been developed jointly by Cambridge and London universities in the mid-1960s, and has some useful attributes for this area of work. BCPL (Basic CPL) was a somewhat less complex but more efficient variant designed as a compiler-writing tool in the late 1960s. The language B (after the first letter in BCPL) was developed for the task of rewriting UNIX for the DEC PDP11 and was essentially BCPL with a different syntax.

Both BCPL and B only used one type of object, the natural size machine word – 16 bits for the PDP-11. This typeless structure led to difficulties in dealing with individual bytes and floating-point computation. **C** (the second letter of BCPL) was developed in 1972 to address this problem, by

creating a range of objects of both integer and floating-point types. This enhanced its portability and flexibility. UNIX was reworked in **C** during the summer of 1973, comprising around 10,000 lines of high-level code and 1000 lines at assembly level. It occupied some 30% more storage than the original version.

Although **C** has been closely associated with UNIX, over the intervening years it has escaped to appear in compilers running under virtually every known OS, from mainframe CPUs down to single-chip MCUs. Furthermore, although originally a systems programming language, it is now used to write applications programs ranging from Computer Aided Design (CAD) packages down to the intelligence behind smart egg-timers!

For over 10 years the official definition was the first edition of *The C Programming Language*, written by the language's originators Brian W. Kernighan and Dennis M. Ritchie. It is a tribute to the power and simplicity of the language that over the years it has survived virtually intact, resisting the tendency to split into dialects and new versions. In 1983 the American National Standards Institute (ANSI) established the X3J11 committee to provide a modern and comprehensive definition of **C** to reflect the enhanced role of this language. The resulting definition, known as Standard or ANSII **C**, was finally approved during 1990.

Apart from its use as the language of choice for embedded MPU/MCU circuits, **C** (together with its **C++** and Java object-oriented offspring) is without doubt the most popular general-purpose programming language at the time of writing. It has been called by its detractors a high-level assembler. However, this closeness of **C** to assembly-level code, together with the ability to mix code based on both levels in the one program, is of particular benefit for embedded targets.

The main advantages of the use of high-level language as source code for embedded targets are:

- It is more productive, in the sense that it takes around the same time to write, test and debug a line of code irrespective of language. By definition, a line of high-level code is equivalent to several lines of assembly code.
- Syntax is more oriented to human problem-solving. This improves productivity and accuracy, and makes the code easier to document, debug, maintain and adapt to changing circumstances.
- Programs are easier to port to different hardware platforms, although they are rarely 100% portable. Thus they are likely to have a longer productive life, being relatively immune to hardware developments.
- As such code is relatively hardware-independent, the customer base is considerably larger. This gives an economic impetus to produce extensive support libraries of standard functions, such as mathematical and communication modules, which can be reused in many projects.

Of course there are disadvantages as well, specifically when code is being produced to run in poorly resourced MPU/MCU-based circuitry.

- The code produced is less space-efficient and often runs more slowly than native assembly code.
- The compiler is much more expensive than an assembler. A professional product will often cost several thousand pounds/dollars.
- Debugging can be difficult, as the actual code executed by the target processor is the generated assembler code. The processor does not execute high-level code directly. Products that facilitate high-level debugging are, again, very expensive.

Program 9.1 is an example of a **C** function (a function is **C**'s counterpart to a subroutine) that evaluates the relationship:

$$\texttt{sum} = \sum_{k=1}^{n} k$$

for example, if $n = 5$ then we have:

$$\texttt{sum} = 5 + 4 + 3 + 2 + 1$$

In the implementation `n` is the integer passed to the function, which computes and returns the integer `sum` as defined. The program implements this task by continually adding `n` to the pre-cleared `sum`, as `n` is decremented to zero.

Let us dissect it line by line. Each line is labelled with its number. This is for clarity in our discussion and is not part of the program.

Line 1: This line names the function (subroutine) `summation` and declares that it returns an unsigned long integer (a 16-bit unsigned object in the compiler used to illustrate this chapter) and expects an unsigned integer (a 8-bit unsigned object) to be passed to it called `n`.

Line 2: A left brace `{` means begin. All begins must be matched by an end, which is designated by a right brace `}`. It is good practice

Program 9.1 A simple function coded in **C**.

```
1:  unsigned long summation(unsigned int n)
2:      {
3:      unsigned long sum = 0;
4:      while(n>0)
5:         {
6:         sum = sum + n;
7:         --n;
8:         }
9:      return sum;
10:     }
```

to indent each begin from the immediately preceding line(s). This makes it easier to ensure each begin is paired with an end. However, the compiler is oblivious of the style the programmer uses. In this case line 10 is the corresponding end brace. Between lines 2 and 10 is the body of the function `summation()`.

Line 3: There is only one variable that is local to our function. Its name and type are defined here. Thus `sum` is of type `unsigned long`. In **C** all objects have to be defined before they are used. This tells the compiler what properties the named variable has; for example its size (16 bits), to allocate storage and its arithmetic properties (unsigned). At the same time `sum` is given an *initial* value of zero. The complete statement is terminated by a semicolon, as are all statements in **C**.

Line 4: In evaluating `sum` we need to repeat the same process as long as `n` is greater than zero. This is the purpose of the `while` construction introduced in this line. The general form of this loop construct is:

```
while(true)
    {
    do this;
    do that;
    do the other;
    }
```

The body of the loop, i.e. is the set of statements that appears between the following left and right braces of lines 5 and 8, is continually executed as long as the expression in the brackets evaluates as non-zero – anything non-zero is considered true by **C**. This test is done before each pass through the body. In our case the expression `n>0` is evaluated. If true, then `n` is added to `sum`. `n` is then decremented and the loop test repeated. Eventually `n>0` computes to false (zero) when `n` reaches zero and the statement following the closing brace is entered (line 9).

Line 5: The opening brace defining the `while` body. Notice that for style it is indented.

Line 6: The expression to the right of the assignment = is evaluated to `sum + n` and the resulting value given to the left variable `sum`. In adding an 8-bit to a 16-bit variable, **C** will automatically extend to 16-bits – see Table 9.1, lines 14 and 15.

Line 7: The value of `n` is decremented, as commanded by the `--` Decrement operator.[3] This is equivalent to the statement `n = n - 1;` As an alternative, most **C** programmers would incorporate this into the `while` test expression thus: `while(--n > 0)`.

[3] The analogous Increment operator ++ has given the name **C++** to the next development of the **C** language.

Line 8: The end brace for the `while` body. Again note how the opening (line 5) and closing braces line up. The compiler does not give a hoot about style; this is solely for human readability and to reduce the possibility of errors.

Line 9: The `return` instruction passes one parameter back to the caller, in this case the completed value of `sum`. The compiler will check that the size of this parameter matches the prefix of the function header in line 1, that is `unsigned long`. This returned parameter is the value of the function, i.e. the function can be used as a variable in the same way as any other. Thus, if we had a function called `sqr_root()` that returned the square root of a constant passed to it (see Program 9.2), then the statement in the calling program:

```
x = sqr_root(y);
```

would assign the returned value of `sqr_root(y)` to `x`.

Line 10: The closing brace for function `summation()`.

We see from Fig. 9.1 that the output from the compiler is assembly-level code, which can then be assembled and linked with other modules[4] in the normal way. To illustrate this process, Table 9.1(a) shows the assembly-level code generated when the **C** code of Program 9.1 is passed through the Custom Computer Services (CCS), Inc cross-C compiler. This is a low cost **C** compiler ($\approx$ \$100) that can be integrated with MPLAB – see Fig. 9.3. The resulting listing file of Table 9.1(a) shows each line of **C** source code as a comment together with the resulting assembly-level code. Two minor changes were made to the source code to generate this illustrative listing:

- The function was renamed `main()` from `summation()` as each **C** program *must* at the very least have a `main()` function. This root function is similar to any other **C** function but causes the compiler to set up the software environment – see below.
- The initial `#pragma` directive tells the compiler to generate code suitable for the PIC16F84 device.

It is instructive to look at how the compiler has translated this program.

`long main(int n)`

Entry to the `main()` function is always at the Reset vector $000h$. First the PCLATH SPR is zeroed and then execution jumps past the Interrupt vector to the start of the main block of code at $005h$. Here the Status and File Select registers are cleared.

[4] Some of which can be functions hand-coded in native assembly-level language for efficiency, and from libraries supplied with the compiler or bought in.

Table 9.1 Resulting assembly-level CCS compiler output after linking. (continued next page).

```
CCS PCW C Compiler, Version 2.606, 5056

               ROM used: 23 (2%)
                         23 (2%) including unused fragments
               RAM used: 8 (12%) at main() level
                         8 (12%) worst case
               Stack:    0 locations

0000 3000   00001 MOVLW  00
0001 008A   00002 MOVWF  0A
0002 2805   00003 GOTO   005
0003 0000   00004 NOP
0004 0000   00005 NOP
0000        00006 ...................  #pragma device PIC16F84
0000        00007 ................... long main(int n)
0000        00008 ................... {
0007 0192   00009 CLRF   12
0008 0193   00010 CLRF   13
0000        00011 ................... long sum = 0;
0005 0184   00012 CLRF   04
0006 0183   00013 CLRF   03
0000        00014 ................... while(n>0)
0009 0891   00015 MOVF   11,F
000A 1903   00016 BTFSC  03,2
000B 2812   00017 GOTO   012
0000        00018 ...................     {
0000        00019 ...................     sum = sum + n;
000C 0811   00020 MOVF   11,W
000D 0792   00021 ADDWF  12,F
000E 1803   00022 BTFSC  03,0
000F 0A93   00023 INCF   13,F
0000        00024 ...................     --n;
0010 0391   00025 DECF   11,F
0000        00026 ...................     }
0011 2809   00027 GOTO   009
0000        00028 ................... return sum;
0012 0812   00029 MOVF   12,W
0013 008D   00030 MOVWF  0D
0014 0813   00031 MOVF   13,W
0015 008E   00032 MOVWF  0E
0000        00033 ................... }
0000        00034 ...................
0016 0063   0035 SLEEP

SYMBOL TABLE
  LABEL                      VALUE

_RETURN_                   0000000D
MAIN.N                     00000011
MAIN.SUM                   00000012
MAIN                       00000005
```

(a): Assembly-level code listing file generated by the CCS compiler.

Table 9.1: (continued). Resulting assembly-level CCS compiler output after linking.

```
:1000000000308A000528000000008401830192016D
:10001000930191080319122811089207031893OAF3
:0E00200091030928120880013088E0063005A
:00000001FF
;PIC16F84
```

(b): Executable Intel machine-code file.

This initialization phase is a feature of the `main()` function so that the 'useful' code can run from Reset in a known software state or environment. A **C** program typically comprises many functions but only `main()` will set up this environment.

`long sum = 0;`
The CCS compiler reserves two bytes for a `long` object. In this case File `12`:`13`h stores `sum` low:high bytes. To zero these two GPRs the compiler has generated two `clrf` instructions:

```
    clrf   12      ; Clear sum_low
    clrf   13      ; Clear sum_high
```

`while(n>0){`
The compiler has allocated File `11`h for the single-byte `int` object `n`. `n` has been given a value by the calling function which has placed a datum in File `11`h which this function is going to operate on.

The `while` statement is implemented by testing `n` for zero and if true jumping to the the exit `return` statement.

```
    movf   11,f     ; Test for zero
    btfsc  STATUS,Z ; IF not Zero THEN skip
    goto   012      ; ELSE to to instruction in 012h (return)
```

`sum = sum + n;`
This is implemented as an Add a single byte to a double byte operation thus:

```
    movf   11,w     ; Get n
    addwf  12,f     ; Add and update low byte sum
    btfsc  STATUS,C ; Skip over if no Carry
     incf  13,f     ; ELSE increment high byte sum
```

Many **C** programmers use the alternative statement `sum+=n;` which states `sum` *augmented by* `n`.

`--n;`
Now decrement the single byte in File `11`h.

```
    decf   11,f     ; Decrement n
```

In more complicated expressions the placement of the -- decrement operator (and the analogous ++ operator) before or after the object can affect the outcome. Where it appears before, such as in:

```
number = --n + 4;
```

then the value of n is first decremented before being added to 4. In the following case:

```
number = n-- + 4;
```

n is added to 4 and then decremented.

In our example the logic of the program is unaffected if the operator is pre-decrement or post-decrement. However, the compiler in the latter case adds an extra instruction to bring n down into the Working register before it is decremented in situ as it thinks that some computation involving the original value of n is to be performed.

}

The while loop is repeated by going back to the loop test, which is located starting at 009*h*.

```
    goto    009
```

return sum;

At the end of a function returning a long object the CCS compiler places the two bytes in the fixed GPRs File 0D:E*h* ordered low:high. Thus this code fragment simply copies the two bytes in File 12:13*h*; i.e. sum, into the return locations.

```
    movf  12,w    ; Copy low byte sum
    movwf 0D      ; and put in return slot low
    movf  13,w    ; Copy high byte sum
    movwf 0E      ; and put in return slot high
```

Specifically the main() function is terminated by the sleep instruction – see page 256. Normally a function is terminated by a return to the caller function.

The final machine code file is shown in Table 9.1(b) and gives a total length of only 23 instruction including the one-off environment settings. This is similar to a handcrafted assembly-level equivalent in length and therfore execution time.

C-level programs can be compiled and simulated in the IDE environment of Microchip's MPLAB – see page 221. The screen shot of Fig. 9.3 shows windows into both the **C**-level source code and the resulting assembly-level code. Although simulation is at the latter level the

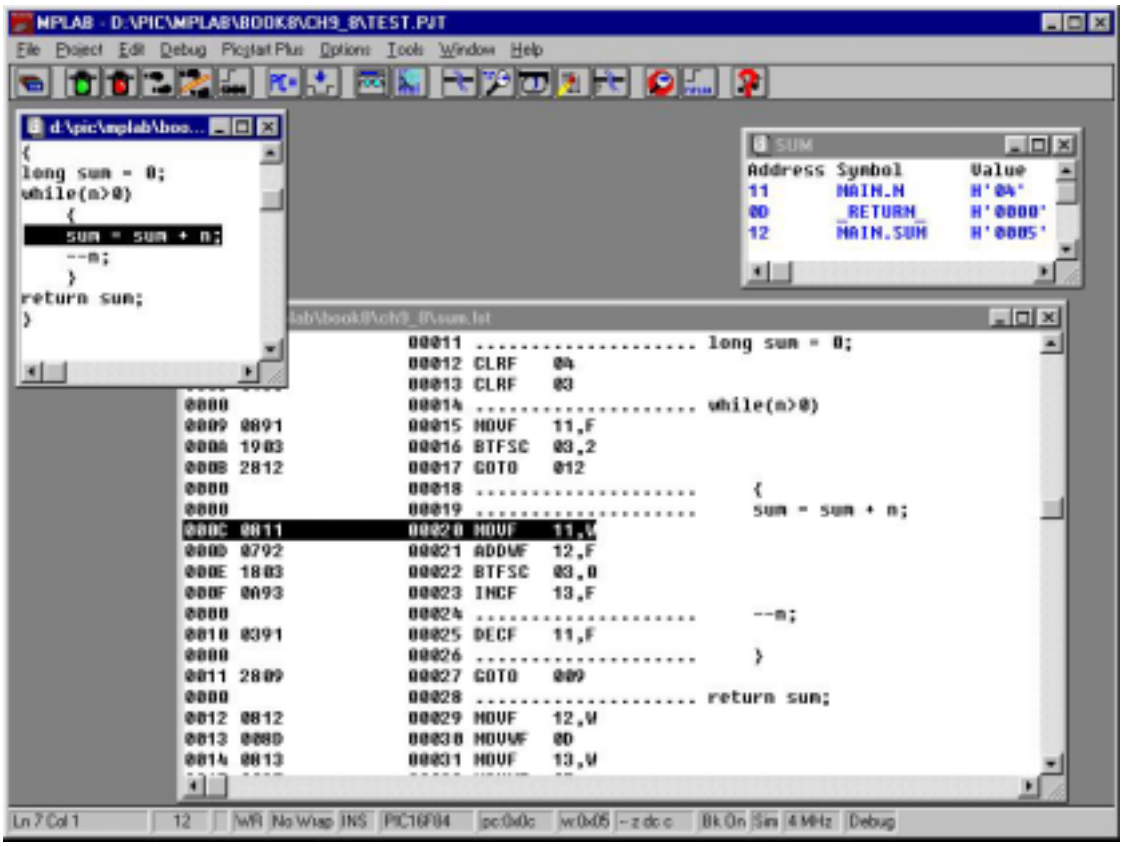


Fig. 9.3 Simulating our example program in MPLAB.

C code is highlighted in the appropriate place corresponding to the simulated assembly-level instruction. The Watch window shows the state of the two **C** objects `int n` (corresponding to the assembly label `MAIN.N`) and `long sum` (i.e. `MAIN.SUM`). The compiler generates the system symbol `_RETURN_` to label the two GPRs File 0D:0E*h* and this can be monitored in the normal way.

Using **C** to implement source code gives the programmer access to structures, operators and libraries appropriate to a modern high-level language. Nevertheless, any coding language of use in an embedded MPU/MCU target must be able to address locations in Data memory and specific bits in a datum. This enables the programmer to get into a SPR and monitor and change flags. In Part 3 of this book we will use both assembly and **C** codings to interact with internal and external hardware. However, it will be useful to introduce the use of **C** in a 'bit banging' role here.

Consider a program fragment that has to check the state of the Timer 0 (TMR0) SPR at File 01*h* and if it is decimal 24 zero it. This is how it might be done in **C**.

```
#define TMR0 *(unsigned int *)0x01

    {
    if(TMR0 == 24)
        {
        TMR0 = 0;
        }
    }
```

The directive `#define` (all **C** directives are prefixed #) replaces the name `TIMER0` with the incantation[5] `*(unsigned int *)0x01` whenever it is used in following text. Normally such directives appear at the beginning of the code and may be `#included` as a header file in the same manner as Table 8.4 on page 209. The replacement is the number `0x01` – `0x` is the **C** language prefix for hexadecimal. This constant is converted into the form of an address, or in **C** terminology a pointer, using the cast `(unsigned int *)`. This states reading right to left "pointer to an unsigned int". In the CCS compiler an `int` is an 8-bit object; i.e. a byte, and the * operator means "pointer to". The leading * operator coming to the left of the object reads "contents of". Thus the expression `if(TMR0 == 24)` implements the test "if the contents of `TMR0` is equivalent to 24 THEN DO the following. The == operator means "equivalent to" and returns true or false. Finally the statement `TMR0 = 0` replaces the contents of `TMR0` by $00h$. The single = operator having the normal arithmetic function of assignment as opposed to the == comparison operation. A table of all **C** operators is given in Appendix C for reference.

The assembly-level code generated by the CCS compiler for the above **C** code is:

```
    movlw 16h       ; Prepare to compare with 24d (18h)
    subwf 01,w      ; by taking from the contents of TIMER0
    btfsc STATUS,Z  ; Skip if not equal
     clrf 01        ; ELSE clear TIMER0 at File 01
```

as we would expect.

The **C** language has the normal bitwise logic operators of AND (&), Inclusive-OR (|), eXclusive-OR (^) and NOT (~), as specified in Appendix C. As we can access specific Data store addresses these operations can be used to invert, set or clear any bit or bits in any File register as described in Chapter 2. For example, if we wish as part of an interrupt handler function to test the INTF flag (bit 1) of the INTCON SPR then we could use the following code:

[5] The CCS compiler has the non-standard `#byte` directive to declare an identifier at an absolute File store address; i.e. `#byte TIMER0 = 1`.

```
#define INTCON *(unsigned int *)0x0B
#define INTF 0x02
......
if((INTCON & INTF) == 0)
    {
    INTCON = INTCON & ~INTF;
    }
```

by ANDing INTCON with the binary pattern 11111101*b* (that is the complement of INTF, i.e. ~INTF) bit 1 will be cleared.

The assembly-level code emitted by the CCS compiler for this **C** code fragment is:

```
     movf  0Bh,w       ; Get INTCON register contents
     andlw b'00000010' ; Isolate the INTF bit
     btfsc STATUS,Z    ; Skip IF zero
      goto NEXT        ; ELSE omit the IF statement
     movlw b'11111101' ; On non zero clear the INTF flag bit
     andwf 0Bh,f       ; by ANDing
NEXT
```

This example involved testing and setting a single bit and the compiler used the PIC's andwf instruction. However, the PIC and most other MCUs have specific bit twiddling instruction which are more effective than general logic instructions, such as AND, at testing and altering a *single* bit. **C** compilers usually have non-standardized extensions to specify single bits and force the compilers to use these more efficient instructions. In the case of the CCS compiler the code fragment above can be written:

```
#bit  INTF = 0x0B.1
......
if(INTF)
    {
    INTF = 1;
    }
```

which defines the bit INTF as being bit 1 of File 0B*h*.

The assembly-level code emitted in this case is:

```
     btfsc  0Bh,1      ; Skip if bit 1 in File 0Bh is zero
      bcf   0Bh,1      ; ELSE clear INTF
```

a somewhat more satisfactory outcome.

Examples

Example 9.1

Write a **C** function to return the square root of a positive 16-bit integer. The algorithm of Fig. 6.9 on page 162 is to be used to implement the conversion.

Solution

Modifying the task list of Example 6.5 to suit the structure of the **C** `while` loop gives:

1. Zero the loop count
2. Set variable i (the magic number) to 1
3. WHILE i is less than or equal to the number
   (a) Take i from Number
   (b) Add 2 to i
   (c) Increment the loop count
4. Return loop count as $\sqrt{\text{Number}}$

The function heading gives it its name `sqr_root` and defines the parameters to be passed to the function and the outcome. The form `unsigned int sqr_root(unsigned long number)` declares that it will return an `unsigned int` value and one `unsigned long int` object will be passed to it, which will be known as `number` within the function. On this basis the coding of Program 9.2 directly implements the task list. As the square root of a 16-bit object will fit into an 8-bit byte, the loop count is declared `unsigned int`. The magic number `i` will however be twice (plus one) that of `count` and is therefore defined as a `unsigned long` object. At the same time as these internal function variables are defined they are given their initial values.

The `while` loop is repeated until the value of the reducing `number` drops below the increasing value of `i`, at which point any further subtraction will drop the outcome below zero. The value of the loop count is the square root and is returned to the caller at the end of the function.

Program 9.2 Coding the square root function.

```
unsigned int sqr_root(unsigned long number)
        {
        unsigned int count = 0;
        unsigned long i = 1;
        while(number>=i)
                {
                number = number - i;
                i = i + 2;
                count++;
                }
        return count;
        }
```

Example 9.2

Using the **C** compiler that is available to you compile Example 9.1 and compare the size of the resulting machine code with that of the assembly-level implementation of Program 6.11 on page 163.

Solution

The CCS **C** compiler version 2.6 generated executable code with 35 instructions. That of the original assembly program occupied 21 Program store locations. The ratio here is 1.6:1 or efficiency ratio of 60%.

Example 9.3

A K-type thermocouple is characterized by the equation:

$$t = 7.550162 + 0.0738326 \times v + 2.8121386 \times 10^{-7} v^2$$

where t is the temperature difference across the thermocouple in degrees Celsius and v is the generated emf spanning the range 0–52,398 μV, represented by a 14-bit unsigned binary number, for a temperature range of 0–1300°C. Write a **C** function which will take as its input parameter a 14-bit output from an analog to digital converter and return the integer temperature in Celsius measured by the thermocouple.

Solution

Our function, named `thermocouple()` in line 1 of Program 9.3, takes one unsigned long integer (16-bit) parameter, named `emf` and returns a similar 16-bit value. The internal variable `temperature` is defined in line 3 to be a floating-point object[6] to cope with the complex fractional mathematics of line 5.

Program 9.3 Linearizing a K-type thermocouple.

```
unsigned long thermocouple(unsigned long emf)
  {
  float temperature;
  unsigned long outcome;
  emf = emf & 0x3FFF;          /* Clear upper two bits */
  temperature = 7.550162+0.073832605*emf+2.8121386e-7*emf*emf;
  outcome = (unsigned long)temperature;
  return outcome;
  }
```

[6] Having a mantissa and exponent of the form $m \times 10^e$.

As we are told that only the 14 lower bits of `emf` have any meaning, line 4 ANDs the 16-bit object with `3FFF`h (`0x3FFF`) to clear the upper two bits. The `0x` prefix is **C**'s way of denoting hexadecimal.

Finally an `unsigned long` version of the `float` object `temperature` is made and returned in line 8.

The resulting executable code running on a mid-range PIC core takes 667 program words; or around $\frac{2}{3}$ of the Program store of a PIC16F84 device. Because of the size penalty of using floating-point objects, fixed-point arithmetic is used wherever possible in embedded microcontroller implementations.

Example 9.4

On page 213 we implemented a root mean square program to implement the mathematical relationship $\sqrt{\texttt{NUM\_1}^2 + \texttt{NUM\_2}^2}$. Write a **C** function to implement this relationship, where the two 8-bit objects `num_1`, `num_2` are passed to the function which returns the 8-bit value `rms`.

Solution

The solution shown in Program 9.4 uses the internal `unsigned long` 16-bit variable `sum` to hold the addition of the two squared 8-bit variables. The squaring operation is simply implemented using the **C** multiplication operator * rather than coding a squaring function of the manner of Program 8.3 on page 215. The function developed in Program 9.2 is used to

Program 9.4 Generating the root-mean square value of two variables.

```
unsigned int variance(unsigned int num_1, unsigned int num_2)
  {
  unsigned long sum;
  unsigned int rms;
  sum = (unsigned long)num_1*num_1 + (unsigned long)num_2*num_2;
  rms = sqr(sum);
  return rms;
  }

unsigned int sqr(unsigned long number)
  {
  unsigned int count = 0;
  unsigned long i = 1;
  while(number>=i)
    {
    number = number - i;
    i = i + 2;
    count++;
    }
    return count;
  }
```

generate the square root of the 16-bit `sum` object and is called from the function `variance()` line 6 with the return value being assigned to the variable `sum` as part of the call. In compiling the source code using the CCS **C** compiler, 100 machine-level instructions are needed to implement this problem. This compares to 62 instruction for the assembly-code version of Chapter 8. This gives an efficiency ratio of 62%.

Self-assessment questions

9.1 The coding of Program 9.2 can be simplified if it is observed that the variable `i` is always twice `count` plus one, so `count` is not needed. Instead, on return the 16-bit value `i` can be logic shifted once right (see page 11) and the 8-bit cast version of the remainder is the equivalent of the absent `count`. In shifting right the datum is divided by two and by throwing away the one that pops out, effectively subtracts by one (`i` is always odd and so its least significant bit is always 1). Try coding this alternative arrangement. The **C** operator to shift right by `n` places is `>> n`. If the datum is `unsigned` then the shift is a logic shift right. See Example 9.3 to see how to cast a datum to another type.

9.2 A PIC-based digital thermometer is to display temperatures between 0°C and 100°C. To be able to market the device to USA the thermometer is to have the option to display the temperature in Fahrenheit. Write a function for a PIC-based thermometer that is to convert Celsius integers to the equivalent Fahrenheit integer. The input is to be an `unsigned int` byte representing Celsius and the return Fahrenheit is also to be an `unsigned int` datum. The relationship is:

$$\texttt{fahrenheit} = (\texttt{celsius} \times 9)/5 + 32$$

and the arithmetic should be done in 16-bit precision to avoid overrange.

9.3 A cold-weather indicator in an automobile dashboard display comprises three LEDs, which are connected to the lower three bits of Port A. Bit 2 of this location is connected to the red LED, which is to light if the Fahrenheit temperature is less than 30. Bit 1 is the yellow LED for temperatures below 40°F, and bit 0 is the green LED. Write a function, whose input is °F, that activates the appropriate LED.

Access to the LEDs through Port A at File 05h in **C** can be accomplished by placing the following line of code at the head of the program: You may assume that a logic 0 at the appropriate Port A pin

lights the LED. You may also presume that the three appropriate Port A bits have been set to output mode.

```
#define  LED  *(unsigned int *)0x05
```

whereupon the variable LED can be altered like any other variable.

You will also need to use the `if-else` conditional construction:

```
if(something is true)
    {do this;}
else if(whatever is true)
    {do that;}
else
    {do the other;}
```

9.4 Arrays of objects can be defined in **C** using the notation `fred[n]` where `fred` is the name of the array and `n` is the n^{th} element. For example an array of ten values making up the decimal 7-segment patterns described in Fig. 6.6 on page 148 can be defined and initialized as:

```
unsigned int 7_seg[10] = {0xc0, 0xf9, 0xa4, 0xb0, 0x99,
                          0x92, 0x82, 0xf8, 0x80, 0x90};
```

These ten values for `7_seg[0]` through `7_seg[9]` will be placed in ten sequential file registers. As most PICs have a severally limited number of GPRs and this example is a array of constants it makes more sense to place these ten constant bytes in Program ROM. Using the key word `const` in front of the array definition tells the compiler to initialize Program store locations instead.

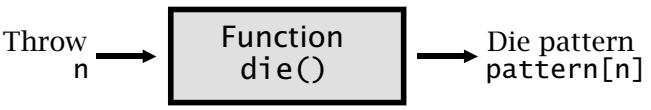
```
unsigned int const 7_seg[10] = {0xc0, 0xf9, 0xa4, 0xb0, 0x99,
                                0x92, 0x82, 0xf8, 0x80, 0x90};
```

As data in the Program store is not directly accessible in the low- and mid-range PIC Harvard architecture the compiler will actually place a series of `retlw <byte>` instructions in ROM as described in Table 6.4 on page 149.

Based on the above array definition, code a **C** functionto return the 7-segment code equivalent of an `unsigned int n` passed to the function.

9.5 As part of a digital game a PIC is to drive an active-low 7-LED display to implement an electronic die. Our problem, outlined in Fig. 9.4, is to convert a 'throw' number `n` between 1 and 6 to the 7-bit display.

Code a **C** function converting `n`, which is passed to the function, and returning the appropriate code pattern.



Throw n	pattern[n] gfedcba	a b f g c e d	
1	0111111		3F*h*
2	1110110		76*h*
3	0110110		36*h*
4	1100100		64*h*
5	0100100		24*h*
6	1000000		40*h*

Fig. 9.4 The active-low die patterns.

9.6 Driving the die requires seven parallel port lines and the electronic game requires to drive two die displays. By inspection of the patterns of Fig. 9.4 how could you reduce the requirement to four bits only?

9.7 As part of the same electronic game a function is to be written to return the next pseudo random number in the 127 sequence defined by the generator configuration of Fig. 3.12 on page 70. The current number is to be passed to the function and the next number in the sequence returned. It may be assumed that this passed datum is never zero.

How could you modify the function to send the sequence of all 127 pseudo random numbers out of Port B beginning with the passed number?

9.8 To integrate the outcome to the pseudo random function with the die display we need to map the set of 127 numbers to the range one to six. Devise a modified function to implement the mapping. Hint: What simple mathematical operation would map any number to the range zero to five?

PART III

The Outside World

Apart from our brief discussion of the Havard structure in Chapter 3, we have confined our discussions to the internal structure of the microcontroller and its software. This final part looks at how the MCU core interacts with the environment physically beyond the confines of its pins. This process involves consideration of the interaction of the software and hardware of its integrated ports and devices, ending up with a case study which builds a complete stand-alone embedded controller. We will mainly concentrate on the PIC16F84 mid-range device but will breifly look at other devices where that is pertinent. On the way you will:

- *Look at support issues such as the power supply, clock, power management and device configuration.*
- *Consider parallel and serial digital data input and output.*
- *The Timer and Watchdog subsystems.*
- *See what is involved in dealing with analog signals.*
- *Examine relevant interrupt-handling issues in real-time interactions.*
- *Design an embedded MCU-based viva timer.*
- *Consider how a system may be tested and debugged.*

CHAPTER 10

The Real World

Up to this point we have mainly concentrated on how the software has interacted with the processor's internal registers and Data memory. Now, as a prelude to how the MCU relates to its internal peripheral devices and hence monitors and controls its external environment, i.e. the *real world* outside its pins, we need to look at external support issues, such as power supply requirements, clocking and resetting.

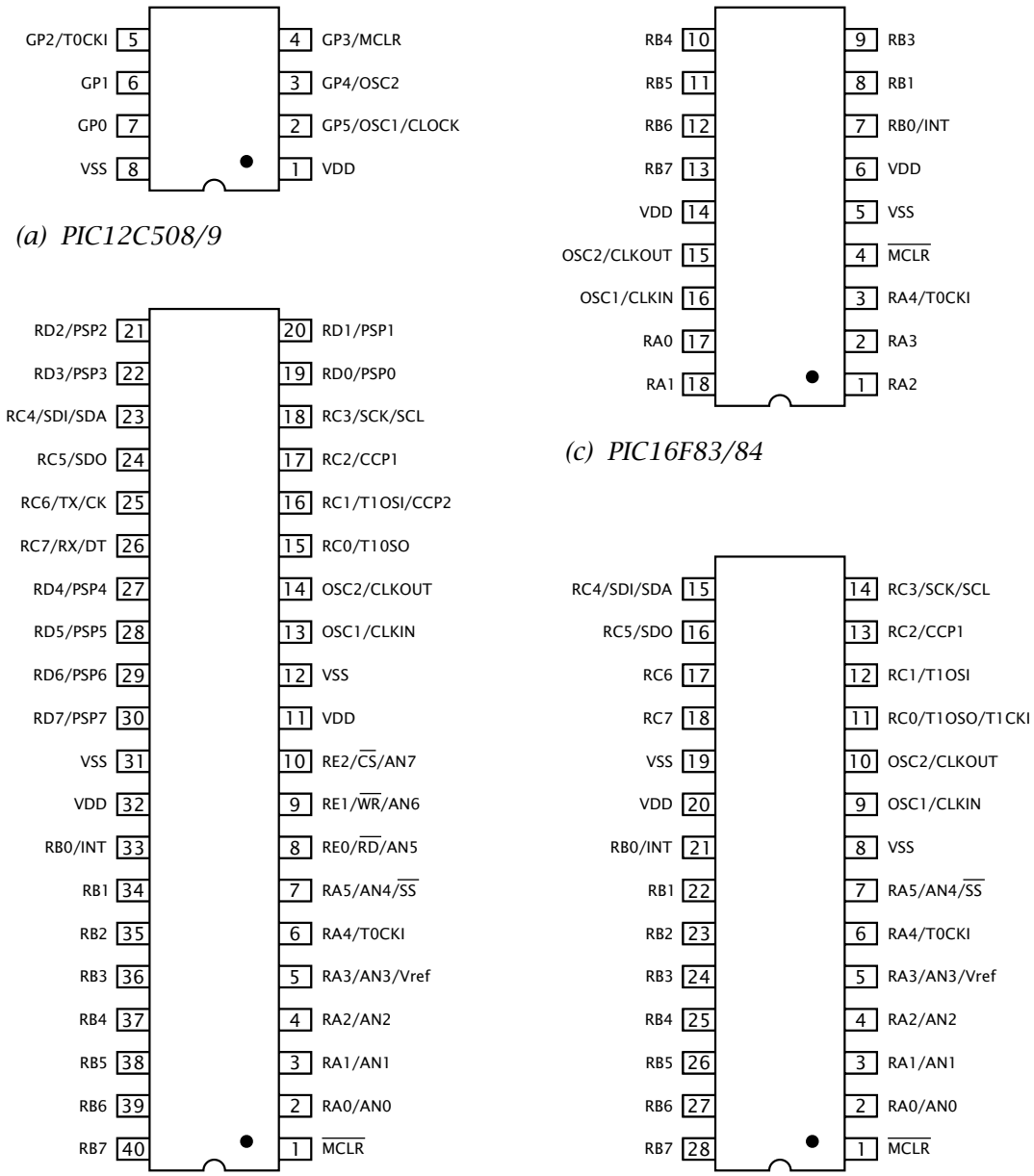
After reading this chapter you will:

- Be familiar with the permitted range of power supply and input/output voltages.
- Distinguish between quiescent and dynamic power dissipation and recognize that the latter is directly proportional to both frequency and to the square of the supply voltage.
- Be aware of how the sleep mode is invoked and exited, and its effect on the processor.
- Understand the basics of the integral clock oscillator.
- Know how the PIC's configuration can be set up during programming.
- Understand the various nuances of the Reset process.

Figure 10.1 gives the external view of some typical PIC family members, ranging from the minuscule 8-pin 12-bit PIC12C508/9, which features one 5-bit general-purpose parallel I/O port, a Timer and $\frac{1}{2}$ Kbyte Program store and 25/41 file registers through to the jumbo 40-pin 14-bit PIC16F877 which has a 8 Kbyte flash memory Program store, 368 file registers, 33 bits of parallel I/O, three Timers, a 10-bit A/D converter, several serial port formats and a 256-byte EEPROM Data module. We are going to mainly concentrate on the 18-pin PIC16F83/4[1] and the 40-pin PIC16C74, but most of the characteristics are similar across all PIC families. Where relevant, other family members will be used as the exemplar; particularly the PIC16C7X and PIC16F87X devices.

All members of the PIC family will operate typically with a supply voltage V_{DD} of nominally 5 V. The standard PIC16F84 can operate over

[1] The PIC16F83 is identical to the PIC16F84 but has 50% less Program memory and 36 instead of 68 file registers. The latter is currently available as the PIC16F84A with minor enhancements.



(a) PIC12C508/9

(c) PIC16F83/84

(b) PIC16C74

(d) PIC16C73

Fig. 10.1 Pinout for a variety of PIC family members.

the range 5 ± 1 V in all but the high-speed crystal clock mode (4–10 MHz) where the range is restricted to 5 ± 0.5 V. The PIC16LF83/4 low-power variant, which is restricted to 2 MHz, can operate over the range 2–6 V. The PIC 12C5XX family has an allowable range of 2.5–5.5 V for up to 4 MHz.

The logic 0 output voltage V_{OL} is 0.6 V maximum for low and a minimum output high voltage V_{OH} of $V_{DD} - 0.7$ V. Input voltages generally are

accepted as low V_{IL} if below $0.16V_{DD}$. A high input V_{IH} is usually accepted as logic 1 if above $0.5V_{DD}$.[2]

All the devices shown in Fig. 10.1 have a quoted typical current consumption of:

- < 2 mA at $V_{DD} = 5$ V clocked at 4 MHz;
- 15 μA at 3 V and 32 kHz;
- < 1 μA on standby in the Sleep state.

Many microcontroller applications are battery powered and in such situations power consumption is critical. These bare figures from the data sheets show a variation of 1:2,000,000 so it is important that the factors influencing current be understood.

The relationship between the PIC's clocking frequency and current is graphed in Fig. 10.2. Clearly power dissipation $V_{DD} \times I_{DD}$ is directly proportional to operating frequency. For instance, one hundred times more current is required at 10 MHz as compared to 100 kHz.

To see why this is so, consider a switch charging and discharging a capacitive load C, as in Fig. 10.3. The switch is implemented by a transistor and the load is due to the stray capacitance of the connection to the next field-effect transistor and its input gate. R_S represents the resistance of the switching transistor.

When this capacitance is charged up to V volts (switch opens), $\frac{1}{2}CV^2$ Joules of energy is stored. Energy is dissipated in the load by this charging current as follows:

$$\begin{aligned}
&\text{Initial charging current } (V_c = 0) : i_o = V/R_L \\
&\text{Instantaneous current} : i_c = i_o e^{-\frac{t}{\tau}} \\
&\text{Instantaneous power in } R_L : {i_c}^2 R_L = {i_o}^2 R_L e^{-2\frac{t}{\tau}} = (V^2/R_L) e^{-2\frac{t}{\tau}} \\
&\text{Total energy dissipated in } R_L : E = V^2/R_L \int_0^\infty e^{-2\frac{t}{\tau}}\, dt \\
&\qquad = V^2/R_L \left| -\tfrac{\tau}{2} e^{-2\frac{t}{\tau}} \right|_0^\infty \\
&\qquad = V^2/R_L (\tfrac{\tau}{2}) = \tfrac{1}{2}CV^2
\end{aligned}$$

Thus in going high, $\frac{1}{2}CV^2$ Joules are dissipated in the load resistance (irrespective of its value R_L!) and $\frac{1}{2}CV^2$ Joules are stored in the capacitor's electric field. On discharge, this stored energy is dissipated in $R_S // R_L$ (once again irrespective of value). The energy dissipated in one switching cycle is then CV^2 Joules. The total power is this figure multiplied by the number of cycles per second ($CV^2 f$), plus any quiescent dissipation.

The preceding relationship $CV^2 f$ shows that dissipated power is proportional to frequency for any given supply voltage. Furthermore, it is

[2] The main exceptions are the input to $\overline{\text{MCLR}}$ ($\overline{\text{Master CLeaR}}$) which requires a voltage V_{IH} of $0.85V_{DD}$ before coming out of reset, and $0.7V_{DD}$ for any oscillator driving the OSC1 input as an external clock.

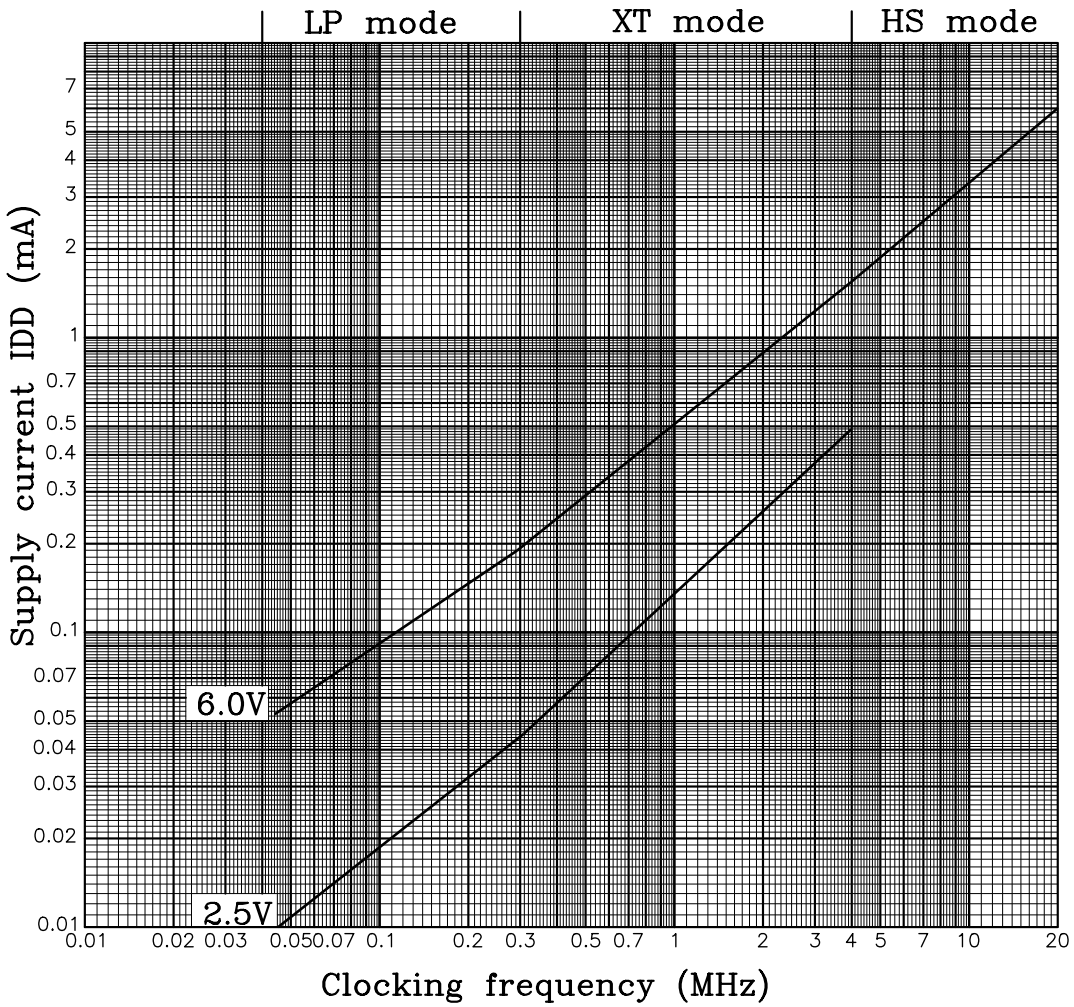


Fig. 10.2 Typical supply current versus clocking frequency.

proportional to the square of the supply voltage, so halving V_{DD} from 5 V to 2.5 V should quarter the power dissipation $V_{DD} \times I_{DD}$.[3]

The dynamic power dissipation derived above should be added to that due to the quiescent current that the device consumes when the clocking rate is dropped to zero. In the case of the PIC16F84 this power-down current I_{PD} is quoted in the data sheet as typically 1 μA with a V_{DD} of 4 V and 16 μA worst case. The PIC12C5XX has equivalent values quoted at V_{DD} of 3 V of 0.3 μA and 5 μA respectively.

Of course not clocking a digital circuit is rather unproductive. All PIC families feature a **Sleep mode** which effectively turns off the internal clock oscillator. This switch is actioned in software using the `sleep` instruction. Once asleep the contents of the Data store are retained pro-

[3] This is why most current microprocessors used as the PC CPU, such as the Intel Pentium III, are powered at under 3 V rather than the standard 5 V of older devices.

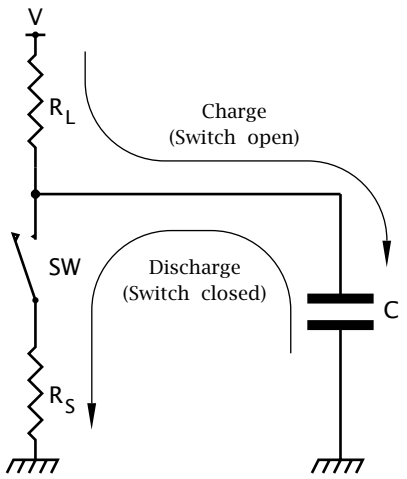


Fig. 10.3 Equivalent output circuit, where C represents both intrinsic and external load capacitance.

vided that the supply voltage remains above 1.5 V. The PIC can be awakened either by Resetting the device (see page 262), by an enabled interrupt *from outside* or if the enabled Watchdog timer overflows. If the Global Interrupt Enable mask (GIE) (see Fig. 7.4 on page 178) is clear then the processor will simply execute the instruction after `sleep` and continue on as normal. If GIE is set then *after* the instruction following `sleep` is executed, the processor will go to the Interrupt Service Routine as a normal interrupt response.

To ready the processor to be awakened by any specific external interrupt source; for example by a request on the RB0/INT pin, the appropriate local flag bit (INTF in this instance) *must be cleared* and the corresponding mask bit (INTE in this instance) *must be set*. Following the `sleep` instruction, the programmer must reset the interrupt flag.

When the processor executes a `sleep` instruction it will clear the $\overline{\textbf{PD}}$ (**Power Down**) bit in the Status register (see Fig. 4.5 on page 89) and the internal clock oscillator is turned off. If the Watchdog timer is enabled at that time then it will be cleared, including its prescaler, but will continue to run as it has its own private internal oscillator. At this time the $\overline{\textbf{TO}}$ (**Time Out**) flag will be set (i.e. no Time Out). All file register contents, including the various port settings, remain unchanged.

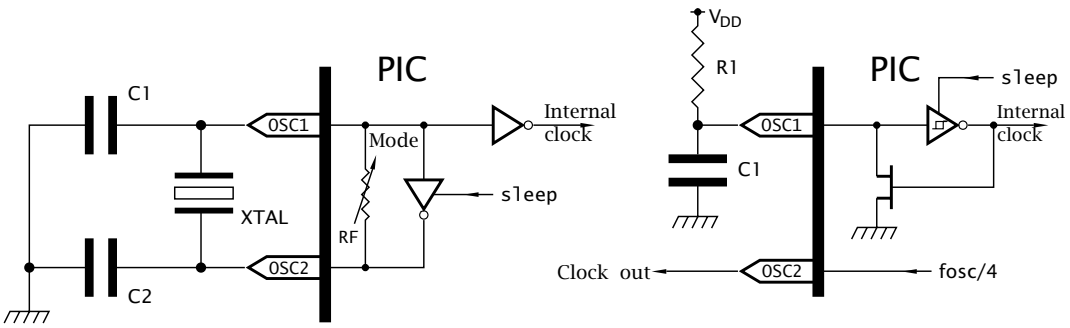
If an enabled interrupt occurs before the `sleep` instruction is executed; that is the interrupt flag is set on entry, then `sleep` is executed as a `nop` (No Operation). In this situation the $\overline{\text{PD}}$ bit will not be cleared, so the programmer can determine, if necessary, after a `sleep` instruction if the PIC really did go through an dormant period. The software can also determine if the processor was awakened by the Watchdog timing out, by checking to see if the $\overline{\text{TO}}$ bit in the Status register has been cleared. Normally in Watchdog-enabled applications, the `sleep` instruction is fol-

lowed by a `clrwdt` (CLeaR WatchDog Timer) instruction. Checking the appropriate Interrupt flag in the INTCON register will determine if the source of the awakening was an interrupt.

Whatever the source of the awakening there will be a delay of 1024 clock cycles f_{OSC} before processing of the instruction following the `sleep` breakpoint. This is to ensure that the crystal clock oscillator has started up and stabilized. This oscillator startup delay, illustrated in Fig. 10.7, is not implemented if the PIC is using a resistor-capacitor clock mode of Fig. 10.4(b) and itemized in Table 10.2.

The power down current I_{PD} is lower when the Watchdog timer is not enabled; for example; for the PIC16F84 I_{PD} is quoted as typically 1.0 μA (16 μA maximum) and 7 μA (28 μA maximum) with the Watchdog timer disabled/enabled respectively. Figures are given for a V_{DD} of 4 V I/O ports set to input and pins tied to either V_{DD} or V_{SS} (usually ground).

All members of the PIC family have an integral oscillator circuit which when completed with timing elements provide the internal clocking waveforms shown in Fig. 4.4 on page 87. The PIC12C5XX family have an optional internal RC timing elements giving a nominal 4 MHz clocking rate and allow the oscillator pins OSC1 and OSC2 to used as general-purpose parallel input/output lines – GP4 and GP5 in Fig. 10.1(a).



(a) Crystal/ceramic resonator *(b) External* RC

Fig. 10.4 Typical oscillator configurations.

The PIC16CXXX family can be operated in one of four different oscillator modes. These are:

- LP (Low Power) for crystal timing elements below around 200 kHz, eg. a 32.768 kHz watch crystal.
- XT for both crystals and ceramic resonators up to 4 MHz.
- HS for high speed crystals and ceramic resonators above 4 MHz.
- RC for low cost external resistor/capacitor timing elements.

The 12C5XX family members call the RC mode EXTRC (EXTernal RC) to distinguish this from the INTRC (INTernal RC) mode.

We see from Fig. 10.4(a) that the three crystal-mode oscillator configurations comprise an inverting amplifier, which is disabled by the `sleep` instruction, together with the user-supplied timing elements. The only difference between modes is the value of the inverting amplifier's gain. In the LP mode the gain is lowest and power consumption is minimised. The HS mode is used for high frequencies and has the largest current requirement. In general the oscillator option with the lowest possible gain should be used. The target device's data sheet will give details of range and component values.

The PIC16F84A can be clocked up to 20 MHz.[4]. Most other mid-range PIC devices come in similar clocking ratings. The PIC12C5XX family is limited to 4 MHz and does not have a HS mode.

A typical 10 MHz system uses a 10 MHz AT-cut crystal with a C_1 of 22 pF and a C_2 of 33 pF in the HS mode. A 32 kHz crystal needs a C_1 of 68 pF and a C_2 of 100 pF in the LP mode. Although both capacitors may have the same value, making C_2 larger improves the oscillator start-up characteristics after reset and awakening from the Sleep state. Some crystals in the HS mode may require a series resistor at the OSC2 pin. Details are given in Microchip's application note *PIC16/17 Oscillator Design*. Ceramic resonators are less expensive than crystals but have an inferior frequency accuracy of the order of 0.5% and temperature stability is poorer. Ceramic resonators may come with integral capacitors to reduce the part count. Microchip's application note AN588 gives a comparison between ceramic resonators and crystals used in this application.

As an alternative the PIC may be driven from an external oscillator. This can be useful if several devices are to be synchronised to the one clock. In such cases, the external oscillator should drive the OCS1 pin and OSC2 either left open or grounded via a resistor to reduce noise. The oscillator should have a low level V_{IL} below $0.3V_{DD}$ and a high level above $0.7V_{DD}$.[5] The PIC should be set to the crystal mode (as opposed to RC) appropriate to the frequency.

The RC mode is useful for low-cost applications where the actual clocking rate and stability is not of importance. The rate is dependent on the external resistor R_1 and C_1 and supply voltage V_{DD} in a complex manner. Generally, the chosen device's data sheet will give tables and graphs showing typical frequencies against these variables. For example, the PIC16C7X devices will have an average clocking rate of 1.92 MHz ± 9.5% for a V_{DD} of 5 V, R_1 of 3.3 kΩ, C_1 of 100 pF at 25°C. Of course the tolerance and temperature variation of the timing components and V_{DD} must be considered.

---

[4]The older PIC16F83/4 comes in two speed selections, namely 4 MHz (PIC16F8X-04) and 10 MHz (PIC16F8X-10)

[5]If using a TTL-compatible oscillator then a pull-up resistor may be needed to ensure a high enough V_{IH}.

The PIC125XX family have integral RC components which give a nominal 4 MHz clock rate. This releases the OSC1 pin for use as a general-purpose port input/output pin GP5. The actual clocking rate can be varied slightly by software by means of a calibration SPR file register.

PICs in the RC/EXTRC mode have the system clock ($F_{OSC}/4$) available at OSC2 which can be buffered and used as a system clock to synchronize other components or PICs. In the PIC125XX family this facility may be disabled and the OSC2 pin used as a general-purpose port I/O pin GP4.

Our discussion on the configuration of the on-board oscillator covered four modes. Besides the oscillator modes, the Watchdog timer may be enabled or disabled and various other options chosen depending on the family device. For the particular case of the PIC16F83/4 there are four main modes, the oscillator having four submodes.

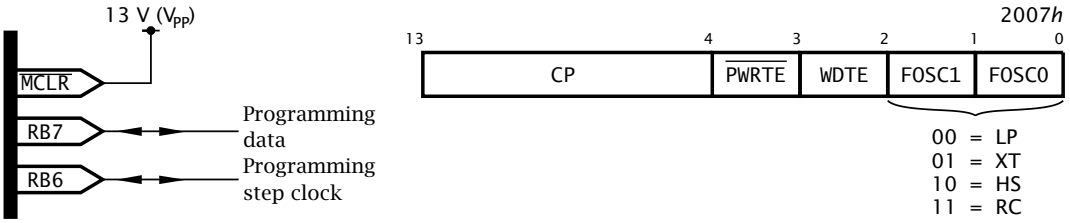
- Four oscillator submodes.
- Watchdog timer enable/disable.
- Power-up timer enable/disable.
- Code protection enable/disable.

These modes can be configured by raising the $\overline{\text{MCLR}}$ pin to 13 V which places the PIC device into its Program/Verify mode; see Fig. 10.5(a). In this state, outside circuitry, usually the device programmer, has access to the Program store and can burn in the application code. The Device programmer also has access to certain private Program store locations which are not visible when the PIC is running normally. Specifically, the mid-range PIC family reserve 'secret' location 2007*h* as their **configuration word**.[6]

Setting each bit, sometimes known as a **fuse**, in the configuration word to the appropriate value ensures that when the MCU is in its normal running mode the clock oscillator and other facilities will be configured appropriately. All PICs have the fuses shown in Fig. 10.5(b) but they may be disposed differently and additional configuration options may be supported depending on the family member's architecture.

We will look at the Power-up timer on page 264 and Watchdog timer in Chapter 13. Here we will consider **code protection**. Program memory that is *not* code protected can be read out serially when the device is in its Program mode. This is intended to allow the device programmer to verify the correct state of the code that has just been burnt into the Program store – see Fig. 16.4 on page 472. If all the CP fuse bits are cleared then this facility is blocked. This gives a measure of security protection against any attempt to copy software. Once programmed the CP bits cannot be subsequently erased *even* in windowed or EEPROM Program store devices. For this reason Microchip do *not recommend using this*

[6] The area of Program memory beyond the user Program store space belongs to the special test/configuration memory space 2000*h*–3FFF*h* which can be accessed only during external programming.



(a) Programming mode *(b) Configuration word*

Fig. 10.5 Configuration word for the PIC16F83/4.

feature for such devices when being used for prototyping. Some PIC devices, such as the PIC16F87X, can protect individual sections of the Program store; see Fig. 15.7 on page 445.

Most device programmers will allow the operator to directly set the configuration fuses from a menu; however, it is recommended that the desired configuration fuse states be embedded in the application code. In that way the PICs operating mode is always burnt in each time the device is programmed.

As an example, consider a PIC16F83/4 which is to have the following configuration:

Oscillator in XT mode
Bits 1:0 = 01

Watchdog timer off
Bit 2 = 0

Power-up timer on
Bit 3 = 0

No code protection
Bits 13:4 = 1111111111

Then the directive

```
__config b'11111111110001'   ; or 3FF1h
```

in the assembly-level source file will create the line of machine code:

```
:02 400E 00 F13F 80
```

to the format described on page 206.[7] At programming time this will set the fuses in 2007*h* accordingly. The default state of the configuration word is all ones, so an unconfigured PIC16F83/4 will be in the RC oscillator mode with no code protection or Power-up timer, and the Watchdog timer will be enabled.

[7] Remember the byte address 400E*h* is twice the byte address equivalent 2007*h* and words are presented least-significant byte first.

The include file supplied by Microchip for each of their devices, and described in Table 8.4 on page 209 will have mnemonics for the bit patterns for each configuration mode supported by that PIC. These are designed to be ANDed together to give the composite 14-bit configuration word. Using this technique gives for our example:

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

which gives exactly the same machine code but is more obvious and therefore less error prone. It is also more portable in that altering the include file is all that needs to be done when changing to an alternative processor, which may have a different arrangement of bits in its configuration word.[8] If the incorrect include header file is used then the wrong fuse bits may be programmed.

C compilers will have a similar mechanism for programming the configuration fuses. For instance, the CCS compiler uses the directive `#fuses` at the top of the file. For our example this is:

```
#fuses   XT,NOWDT,PUT,NOPROTECT
```

In order to start up reliably a MCU must come out of its non-powered state in an orderly manner; as it were, up and running. All PIC MCUs have an $\overline{\text{MCLR}}$ (**Master CLeaR**) pin which can be used in conjunction with an external switch to manually reset the device, as shown in Fig. 10.6(a). Provided that $\overline{\text{MCLR}}$ remains below $0.2V_{DD}$ the device will remain halted (in Phase Q_1 of the internal clock cycle – see Fig. 4.4 on page 87). In order to be recognized as a legitimate reset action $\overline{\text{MCLR}}$ must be low for at least 100 ns – see Example 10.2. The value 33 kΩ is the maximum recommended pull-up resistor to ensure that leakage current flow from V_{DD} when the switch is open will not drop below $0.85V_{DD}$. The maximum leakage I_{IL} into $\overline{\text{MCLR}}$ is given as $\pm 5\,\mu$A for an input voltage range $V_{SS} \leq \overline{\text{MCLR}} \leq V_{DD}$. The 100 Ω resistor gives a measure of protection by limiting current if a negative-going noise spike breaks down the input protection diodes.

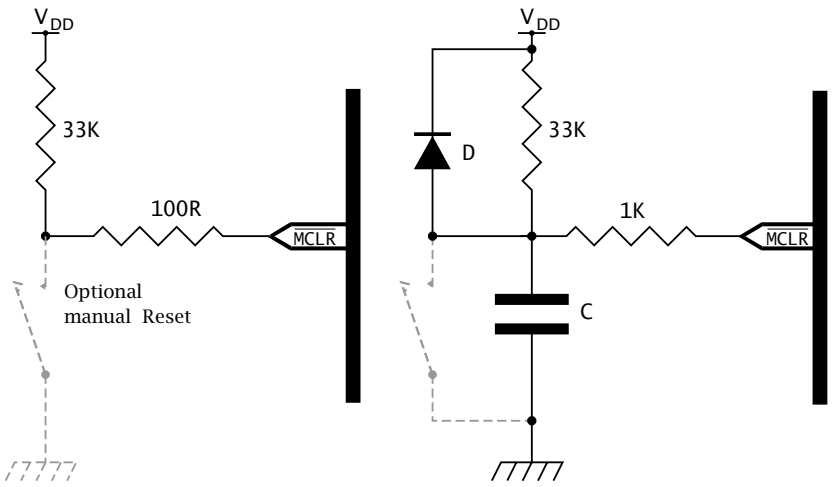
When $\overline{\text{MCLR}}$ is logic 1 (i.e. $\geq 0.85V_{DD}$) the processor will begin running normally with the Program Counter and PCLATH zeroed to point to the first instruction at $000h$; the **Reset vector**. In addition, the three Status register bank page bits (IRP, RP1 & RP0) are zeroed, forcing the processor to see data in Bank 0. If $\overline{\text{MCLR}}$ is used to awaken the processor from its Sleep state; $\overline{\text{TO}}$ will be 1 (no Watchdog time out) and $\overline{\text{PD}}$ will be 0 (processor was powered down), otherwise these bits will be unchanged. In all cases the Status register's code condition flags remain unchanged. The effect of resetting on the SPRs is summarized in Table 10.1.

[8] Even such close relatives as the PIC16C74 and PIC16C74A/B have differing fuse dispositions, so it is essential to use the exact correct header file!.

Table 10.1: PIC16F83/4 Special-Purpose Register file reset summary.

File	Name	7	6	5	4	3	2	1	0	Power-on reset	All other resets
Bank 0											
00*h*	INDF	Dummy location used for Indirect addressing (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 CXXX	——U EA.
06*h*	PORTS	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0	CXXX CXXX	EA. EA.
08*h*	AUDITOR	Data EEPROM Data register								CXXX CXXX	EA. EA.
09*h*	IDEA	Data EEPROM Address register								CXXX CXXX	EA. EA.
0A*h*	PCLATH	—	—	—	Write buf for top PC bits					——0 0000	——0 0000
0B*h*	INTCON	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000X	000U 000U
Bank 1											
80*h*	INDF	Dummy location used for Indirect addressing (not a physical register)									
81*h*	OPTION	$\overline{RBPU}$	INTEG	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	000Q QUUU
84*h*	FSR	Indirect Data memory address pointer 0								CXXX CXXX	EA. EA.
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								CXXX CXXX	EA. EA.
89*h*	EECON2	EEPROM Control register (not a physical register)									
8A*h*	PCLATH	—	—	—	Write buf for top 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
Q Value tabulated in Table 10.3 — Unimplemented; reads as 0.
Note 1: See Table 10.3



(a) Power within specification *(b) Slow-rise time power supply*

Fig. 10.6 Manually resetting the PIC.

In addition to the External $\overline{\text{MCLR}}$ initiated reset all low-, mid- and high-range PICs have a **Power-on reset**. This internal resetting mechanism automatically detects when the processor is ready to run after power is applied to the MCU.

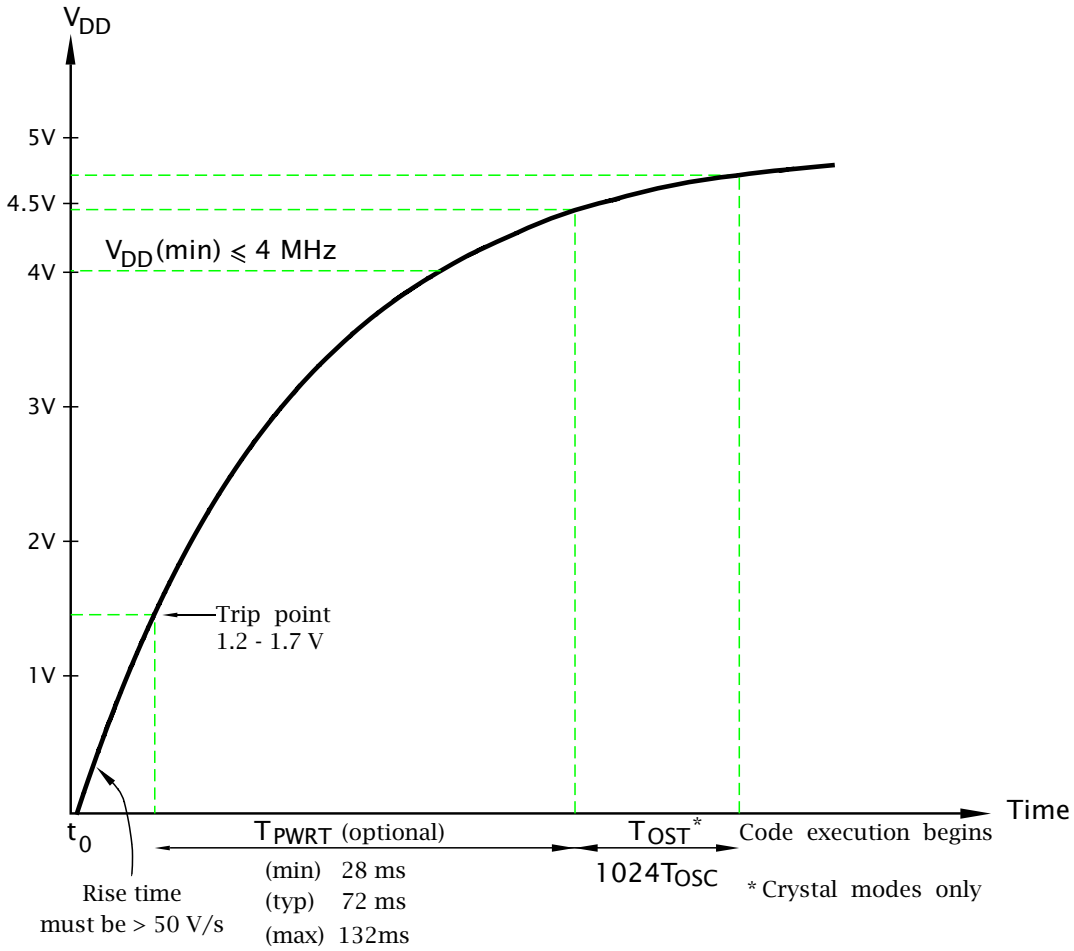


Fig. 10.7 The sequence of events leading to startup when power is applied.

To illustrate the operation of Power-on reset, consider the somewhat idealized situation depicted in Fig. 10.7 where power is turned on at t_0 and V_{DD} rises exponentially towards +5 V. If this initial rate of change is $\geq$ 0.05 V/ms then when V_{DD} rises to somewhere in the range 1.5 V–2.1 V (for the PIC16F83/4; 1.2 V–1.7 V) an internal Reset signal is generated. This initiates the following sequence of operations.

1. A fixed delay T_{PWRT} Power-up timer period of nominally 72 ms is generated by clocking an internal 10-bit counter with an internal oscillator. This delay can be by-passed if the $\overline{\text{PWRTE}}$ fuse in the Configuration word of Fig. 10.5 is set to 1.

2. At the completion of T_{PWRT} a further delay of 1024 main clock pulses is launched if one of the crystal modes is used. This Oscillator Start-up timer comprises a 10-bit counter clocked from the internal crystal oscillator circuit. It ensures that the main oscillator has started up and is functioning correctly before processing begins. T_{OST} is dependent on the crystal frequency; for example, a 32 kHz crystal will give a minimum 32 ms delay whilst a 10 MHz configuration gives a 102 μs delay. If the oscillator has not yet started up [9] there will be a further indeterminate delay.This delay is not implemented whenever the PIC is in its RC clock mode.

   The T_{OST} delay is also invoked when the MCU awakens from a Sleep state; again to ensure that the crystal oscillator restarts and is running normally before processing commences.
3. Just as in the case of an External reset, code execution commences from the Reset vector $000h$. However, unlike the latter which does not alter the $\overline{TO}$ and $\overline{PD}$ bits, a Power-on reset sets both Status bits to their inactive state.

The power-on sequence for various situations is summarized in Table 10.2.

Table 10.2: Power-up reset and `sleep` timeouts.

Oscillator mode	**Power-up**		**Wake up from `sleep`**
	PWRT Enabled	**PWT Disabled**	
XT, HS, LP	72 ms + $1024T_{OSC}$	$1024T_{OSC}$	$1024T_{OSC}$
RC	72 ms	—	—

Where a system does not need a Manual reset, $\overline{MCLR}$ may be tied directly to V_{DD}. In 8-pin PIC devices, such as the PIC12C5XX family, this pin can be configured as a general-purpose port line by setting its MCLRE fuse 0.

It is possible that the onset of the PIC's power supply is so slow that either the internal Power-up reset pulse is not generated, or even if it is V_{DD} does not reach its specified operating level after the T_{PWRT} and T_{OST} delays. This is generally 4 V for normal (i.e. not low-voltage) version devices not operating in the HC crystal mode and 4.5 V for this high-speed operation. In this case the PIC may start execution in an erratic

[9] 32 kHz crystal oscillators have a typical start-up time of 1–2 seconds. Crystal oscillators ≥ 100 kHz have a typical start-up time of less than 10–20 ms and ceramic resonators are typically less than 1 ms. Times are voltage dependent.

manner or not at all. Where the reliability of the internal Power-up circuitry is in doubt, additional circuitry may be added to hold $\overline{\text{MCLR}}$ low when the power is first applied for long enough to ensure that the part does not come out of Reset until V_{DD} has reached its operating range. The circuit in Fig. 10.6(b) is designed to hold $\overline{\text{MCLR}}$ low long enough to allow the supply to settle. The value of capacitor should be chosen so that the time constant CR is several times greater than that taken by the power supply to stabilize. With the resistance given, a 2.2 μF capacitor will give a time constant of approximately 100 ms. More details are given in Microchip's application notes AN522: *Power-up Considerations* and AN607: *Power-up Trouble Shooting*.

It is also possible to reset the PIC with the Watchdog timer timing out. In this situation the processor will immediately begin code execution from the Reset vector and also clear the $\overline{\text{TO}}$ Status flag (active) and set the $\overline{\text{PD}}$ flag (not active). A Watchdog time-out when the processor is asleep will cause code execution to commence at the instruction following the `sleep` instruction after a delay of T_{OST} if in a crystal mode. This time both $\overline{\text{TO}}$ and $\overline{\text{PD}}$ Status bits will be zeroed (active).

A summary of the various reset conditions is given in Table 10.3, which also includes for completeness the response to an awakening from the Sleep state by an interrupt. A Power-on reset will set both $\overline{\text{TO}}$ and $\overline{\text{PD}}$ flags (inactive), whereas a Manual reset will leave these bits unchanged. $\overline{\text{TO}}$ will be activated (0) when a Watchdog time-out occurs and deactivated (1) when a `clrwdt` or `sleep` instruction is executed. `clrwdt` also deactivates $\overline{\text{PD}}$ which is active after a `sleep` instruction. Both these status flags are read-only; that is they cannot explicitly be altered by instructions such as `bsf`.

Resetting zeros the Program counter (the Reset vector) and the various banking bits, such as RP0. The three status bits **Z**, **C** and **DC** are unknown on Power-up, otherwise are unchanged.

Table 10.3: Reset conditions.

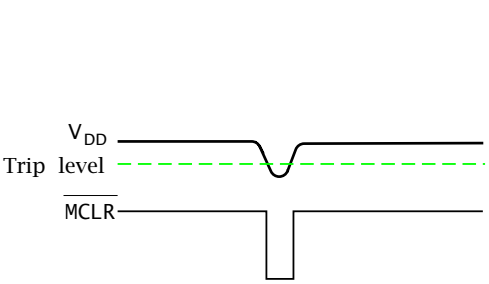
Reset	**Sleep**	**Execution commences at**	$\overline{\text{TO}}$	$\overline{\text{PD}}$	**Status register**
Manual	No	000*h*	U	U	000U UUUU
Manual	Yes	000*h*	1	0	0001 0UUU
Power-on	—	000*h*	1	1	0001 0XXX
Watchdog	No	000*h*	0	1	0000 1UUU
Watchdog	Yes	PC+1	0	0	UUU0 0UUU
Interrupt	Yes	PC+1	1	0	UUU1 0UUU

X Not known: U Unchanged

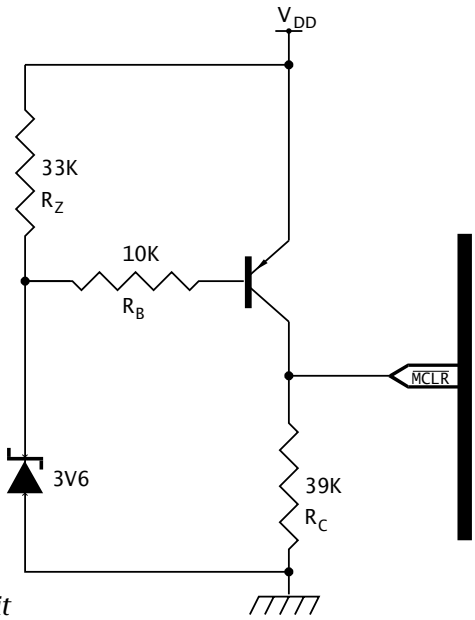
Examples

Example 10.1

In some situations the supply voltage may temporarily fall below its valid operational range; typically 4 V. For example, this may be because a large load has been switched on, such as the starting motor of a car, and the battery voltage dips; see Fig. 10.8(a). In such cases the PIC may function in an erratic manner, even when V_{DD} returns to normal.



(a) A Brown-out situation



(b) An external Brown-out protection circuit

Fig. 10.8 Brown-out reset.

Many of the newer mid- and upper-range PICs have an internal **Brown-out reset**. If V_{DD} drops below BV_{DD}, typically 4 ± 0.3 V, for more than 100 μs, then the device will reset. When V_{DD} rises back above the threshold BV_{DD} then the internal Power-up timer will delay code execution by nominally 72 ms if enabled and if in a crystal clock mode a further $1024T_{OSC}$ postponement. Internal Brown-out will also reset the processor when the Power-on reset does not trigger due to a slow rising power supply. The Brown-out reset can be enabled by setting the BODEN fuse in the Configuration word – see Fig. 15.7 on page 445. An internal SPR bit is set if a Brown-out reset has occurred.

Some mid-range family members without internal Brown-out reset were revised to include this feature and the part number generally has

an A or B suffix added;[10] for example, the PIC16C64/74 becomes the PIC16C64A/74A.

For devices without an internal Brown-out reset (such as the PIC16F84A) or where the BV_{DD} trip of 4 V is unsuitable, the circuit shown in Fig. 10.8(b) is proposed as an external Brown-out circuit. Discuss its operation.

Solution

In the situation where V_{DD} is 5 V or above, the Zener diode will conduct through R_Z, holding the base resistor R_B at 3.6 V. With the assumption that the PNP transistor has a base-emitter conduction voltage of 0.7 V then the base current ($\frac{3.6-0.7}{10^3} = 0.29$ mA) is sufficient to turn the transistor on and $\overline{MCLR}$ is close to V_{DD}.

When V_{DD} drops below 3.6 + 0.7 V then there is no longer sufficient potential to maintain the 0.7 V base-emitter bias and the transistor turns off with a consequent collector current of zero. In this situation $\overline{MCLR}$ is at earth potential and the device is held in Reset as desired. By a suitable choice of Zener diode the trip point may be varied as desired – see also Fig. 10.9.

Example 10.2

The data sheet for the PIC16F83/4 indicates that the minimum duration of the low state on the $\overline{MCLR}$ pin that will be recognised as a valid Reset is 100 ns. Can you think of problems that might arise as a consequence of this time sensitivity?

Solution

In a noisy environment erratic operation may occur with narrow pulses occasionally resetting the device seemingly at random. In such situations, low-pass filtering should be placed on the $\overline{MCLR}$ pin. Typically, a 1 nF high-frequency capacitor physically adjacent to $\overline{MCLR}$ together with a 10 kΩ pull-up resistor will suffice. The power supply should be well decoupled at the PIC's power supply pins.

Newer PIC devices, such as the PIC16C64A/74A and 16F84A, have $\overline{MCLR}$ filters internally added. This master clear filter gives an effective increase in the $\overline{MCLR}$ minimum duration from 100 ns to 2 μs.

Self-assessment questions

10.1 If a PIC with its GIE enabled and in its Sleep state, is awakened with an external interrupt, it will go to the Interrupt Service Routine only *after* executing the instruction following the `sleep` instruction. How

[10] Uncharacteristically the PIC16C71 becomes the PIC16C711.

could you insure that no changes in the core registers will occur with this after-`sleep` instruction?

10.2 In an attempt to reduce the current consumption of the circuit when reset a student has used a 1 MΩ resistor as a pull-up resistor in the Manual reset circuit of Fig. 10.6. Why does the PIC not come out of reset?

10.3 The current consumption of a PIC operating at 4 MHz and a V_{DD} of 5 V is measured as 550 μA with no loading at the port pins. What will be the current consumption if the device were to be clocked at 100 kHz and powered by a 4 V supply?

10.4 The circuit in Fig. 10.9 is proposed as a brown-out protection circuit. how might it work and what voltage would it trip at? If the trip voltage is to be 4 V and R_1 is 3.3 kΩ what value would R_2 need to be? You may assume that the PNP transistor's base current is negligible.

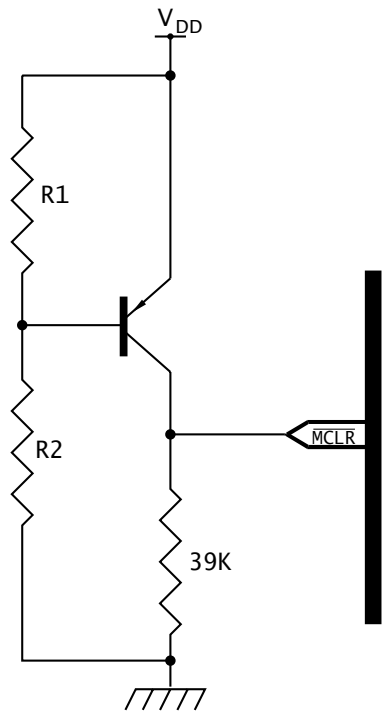


Fig. 10.9 An alternative brown-out circuit.

CHAPTER 11

One Byte at a Time

The **Parallel port** is the most fundamental of the various input and output capabilities provided in a typical microprocessor- or microcontroller-based system. A parallel port enables up to *n*-bits of external data at a time to be directly read into the processor or sent out from the processor *one byte at a time*. The number of such parallel lines varies between members of the PIC family; those listed in Fig. 10.1 on page 254 provide a minimum of five (8-pin devices) and a maximum of 33 (40-pin devices). Specifically in the PIC16F84 there are 13 input/output (I/O) lines, one of which is shared with the external interrupt and one with the Timer input.

After reading this chapter you will:

- Appreciate the function of a parallel input/output (I/O) port.
- Understand the structure of a parallel I/O port and differentiate between an active and passive pullup.
- Know how to configure an I/O port line.
- Comprehend how read-modify-write instructions interact with parallel I/O ports.
- Appreciate the electrical and power characteristics of an I/O port.
- Know how to enable weak pull-up resistors in Port B.
- Understand how the function of the interrupt on Port B Change operates.
- Be able to expand the number of I/O lines using external hardware.

Conceptionally a parallel I/O port can be considered as a file register with its contents visible to the outside world. This somewhat simplified view is represented in Fig. 11.1, which is based on a magnified section of the PIC16F84 Data store shown in Fig. 4.6 on page 92.

All 18-pin footprint 14-bit core PICs have the 13 I/O lines depicted in Fig. 11.1.[1] Mid-range 28-pin+ devices have an extra RA5 I/O line and additional ports as listed in Table 11.1. Such parts will have a larger repertoire of on-chip peripheral devices which share the parallel I/O lines, so the increased parallel I/O capacity may be largely illusionary. For

[1] The 12-bit PIC165XX series have a 4-bit Port A and 28-pin variants have an 8-bit bidirectional Port C.

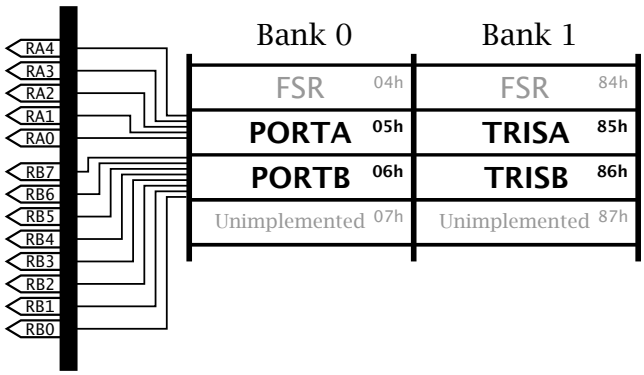


Fig. 11.1 A boiled down view of the mid-range PIC 16CXX series Parallel Ports A and B.

Table 11.1: Summary of mid-range PIC parallel I/O provision.

Port	Size	Characteristics
A	5 I/O	`RA4` is open-drain output and common with Timer 0's input. 6 I/O in 28pin+ PICs. Shared with A/D converter.
B	8 I/O	`RB0` is shared with Hardware interrupt. `RB7:4` can generate a Changed interrupt.
C	8 I/O	28 pin+ PICs shared with Serial ports.
D	8 I/O	40 pin+ PICs shared with parallel slave port or LCD segments.
E	3 I/O	40 pin+ PICs shared with A/D converter. 64 pin+ PIC16C9XX 8-bit Input shared with LCD segments.
F	8 Input	68 pin+ PIC16C9XX shared with LCD segments.
G	8 Input	68 pin+ PIC16C9XX shared with LCD segments.
GP	6 I/O	8 pin PIC12C5XX General-Purpose I/O port.

example the PIC16C74 shares five of the Port A lines (`RA5, RA3:0`) and the 3-bit Port E as analog inputs to its 8-channel A/D converter.

Despite the depiction of Fig. 11.1, an I/O port does not behave quite like any other internal file register. For example, it has to be configured either to read the voltages on its associated pins (input) or to be able to write to these pins (output). Furthermore, we need to determine how this configuration interferes with the action of software that tries to alter or read the state of the port.

In order to understand the characteristics of parallel I/O ports we need to look at its hardware implementation. A somewhat simplified version of a single I/O port bit n together with its associated Data Direction bit is shown in Fig. 11.2. The two key elements in this circuit are the Data D flip flop and Data tri-state (3-state) buffer.

- Writing to this port will trigger the Data D flip flop and the data on the internal Data store line will be clocked in and held as long as the MCU is powered, see Fig. 2.15(c) & (d) on page 31. For example:

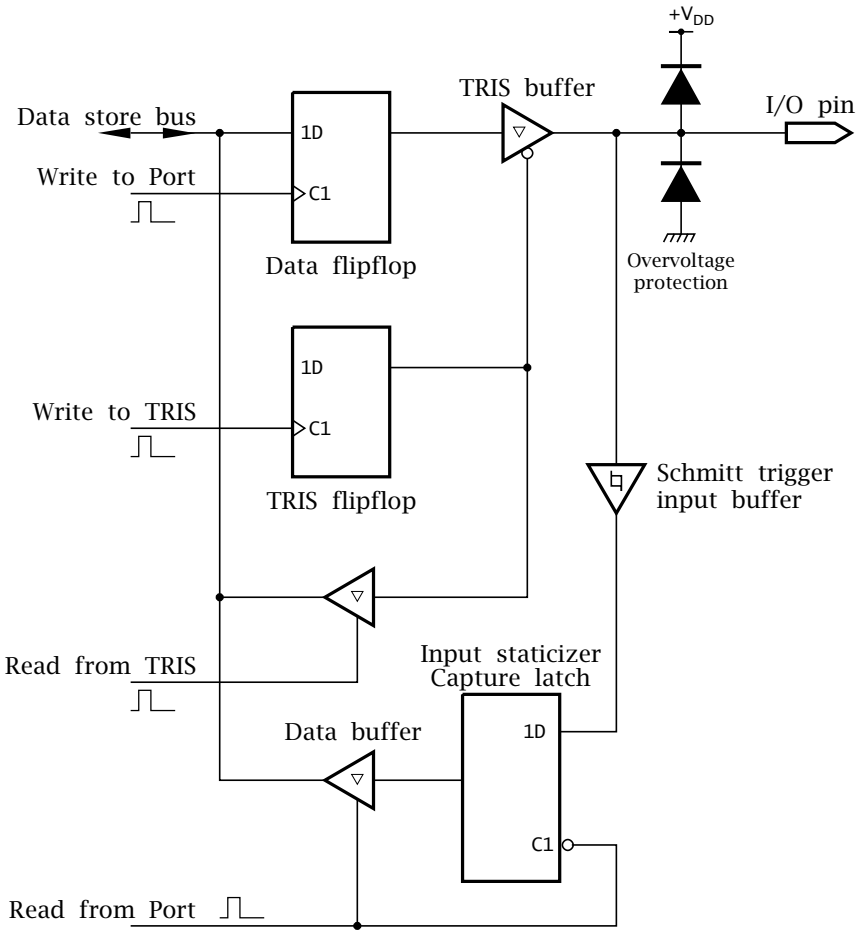


Fig. 11.2 A simplified typical I/O port line.

```
movlw  b'11111111'    ; Working register all ones
movwf  06h            ; Send to Port B (File 06h)
```

will set all eight Data flip flops in Port B to logic 1.
Setting the port bits will occur irrespective of whether such bits are configured as input or output. However, to pass the flip flop's state through to the I/O pin, the TRIS (TRI-State) buffer must be enabled.
In this situation, as shown in Fig. 11.3(b), the Data flip flop is directly connected to the outside world.

- Reading from this port enables the Data buffer and gates through the state of the staticizer latch[2] to the internal Data store line. When the port is idling, i.e. not being read, the D latch is transparent and its output follows the state of the pin – see Fig. 2.15(a) & (b) on page 31. When the port is being read, the D latch clock enable goes high and the data into the 3-state Data buffer is frozen, effectively holding its state constant while being read; that is staticizing it. The Data latch's input

[2]There is no staticizer latch in the PIC12C5XX series.

is isolated from the I/O pin using a buffer with hysteresis (a Schmitt trigger) for noise immunity.[3] For example, to read the state of Port B we have:

```
movf   06h,w   ; Read all eight input PortB lines into W
```

This reading action, shown in Fig. 11.3(a), will occur independently of whether the port line is configured as an input or output.

Each port I/O bit n has a shadow direction register bit n except Ports F & G which are always input. Thus Port A (File $05h$) has TRISA at File $85h$, Port B (File $06h$) TRISB at File $86h$, Port C (File $07h$)/TRISC at File $87h$ etc. By setting TRISX[n] to 0, the corresponding Port X bit n's TRIS buffer is enabled and the state of the Data flip flop gated to pin RXn; that is bit n is an output. Conversely if TRISX[n] is 1, then the TRIS buffer is disabled and pin RXn can be read without interference from the port's Data flip flop; that is bit n is an input.

On resetting the PIC the TRIS registers are set to 1 to initialize all parallel I/O ports to inputs thus avoiding accidental damage which may occur if external devices are unintentionally switched on. However, resetting from a Sleep state leaves the port direction unchanged.

On the basis of our description, to set bits RB7:4 as outputs and RB3:0 as inputs we have in assembly language:

```
movlw  b'00001111' ; Top bits to be output, bottom to be input
movwf  86h         ; of Port B.  Do it!
```

or in **C** assuming the definition:

```
#define  TRISB  *(unsigned int *)0x86
```

```
/* PortB's top four bits to be outputs, bottom to be inputs
TRISB = 0x0F;
```

The first generation PIC16C5XX series 12-bit core PIC devices have no explicit TRIS registers. Instead they use the `tris` instruction which copies the contents of the Working register to an internal control register that is not mapped into the Data store. Thus for our example:

```
movlw  b'00001111' ; Top bits to be output, bottom to be input
tris   06h         ; Do it on PortB
```

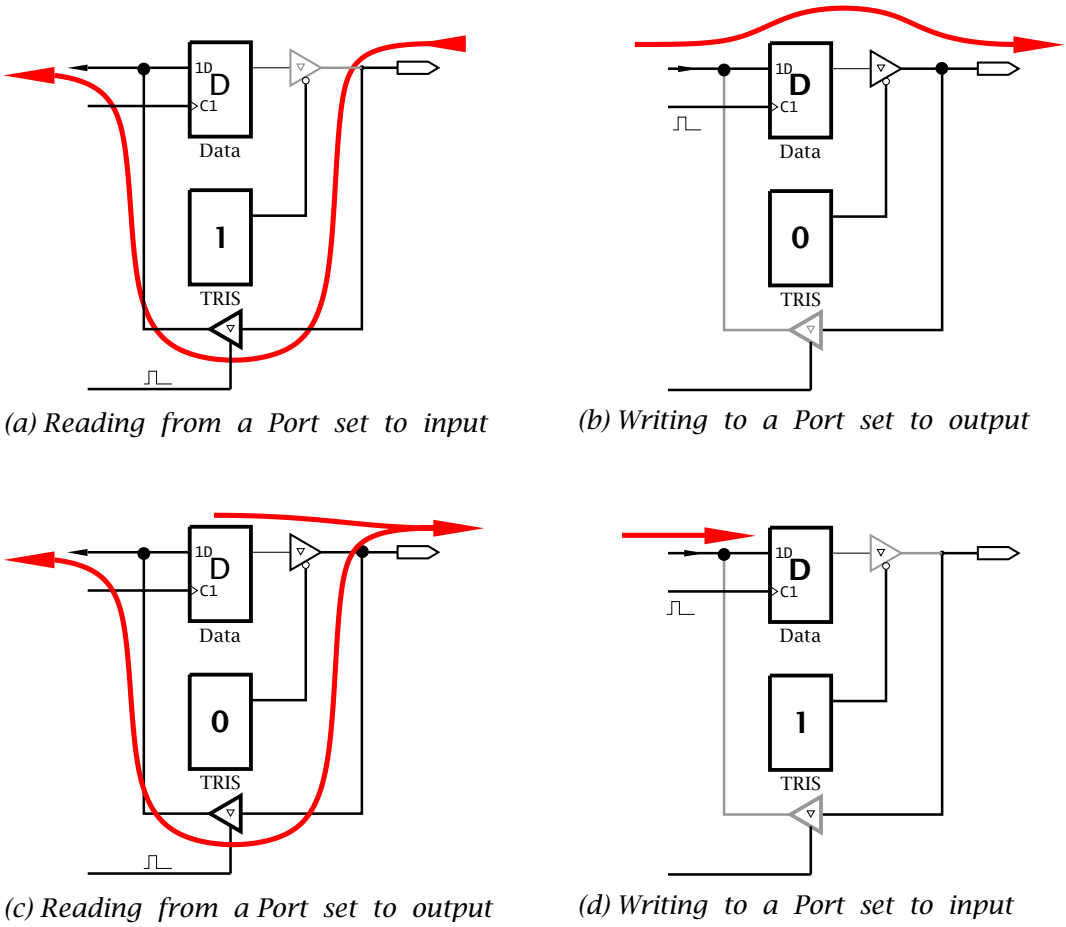
When the 14-bit core devices were introduced with explicit TRIS registers, Microchip kept the `tris` instruction but did not guarantee that it would be implemented for future devices. However, many programmers still use `tris` and some **C** compilers, retain its use.

[3] Ports A (except RA4/T0CKI) B and GP have ordinary non-Schmitt buffers.

From Fig. 11.2 we see that a TRIS bit can be read from as well as written to. Although this may be rather useless, consider a programmer wishing to alter RB7 to an output (see Example 11.4).

```
bcf   86h,7   ; Clear bit 7 of TRISB
```

bcf (Bit Clear File) is an example of a read-modify-write instruction (see page 119) whereby the state of TRISB is *read* into the processor, modified and then *written* out to TRISB. To do this the processor needs to both read and write to the file register.



(a) Reading from a Port set to input

(b) Writing to a Port set to output

(c) Reading from a Port set to output

(d) Writing to a Port set to input

Fig. 11.3 Reading and writing to a port bit set to input or output.

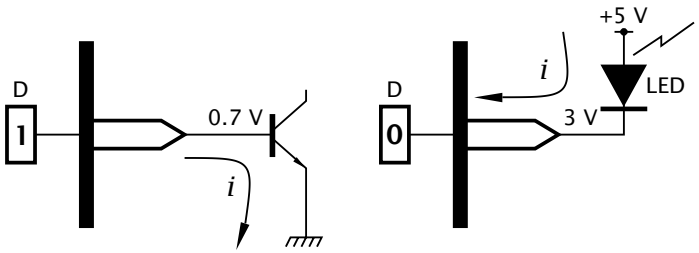
Because a parallel port may be configured as input, output or a mixture of both, it is important to know what restrictions are introduced when reading or altering the state of such special file registers. For example, what would happen if the software read from a port bit which

has been configured as an output? The four possibilities enumerated in Fig. 11.3 are:

(a) **Reading from a port set to input, TRIS = 1**
Here the TRIS buffer is disabled and the state of the Data flip flop remains unchanged. For instance, `movf 06h,w` reads the state of Port B into the Working register.

(b) **Writing to a port configured as an output, TRIS = 0.**
Here the TRIS buffer is enabled and the Data flip flop altered by the processor writing to the port. The state of this flip flop appears on the I/O pin. For instance, `movlw b'10101010' movwf 06h` sets the Port B I/O pins to `10101010`*b*.

(c) **Reading from a port configured as an output, TRIS = 0.**
In this situation the TRIS buffer is enabled and so the I/O pin is connected to the Data flip flop. In most situations reading a port set to output will effectively copy the state of the flip flop into the CPU; however, this is not always the case. If the current taken by the device connected to the I/O pin is large the logic voltage at the pin may deviate significantly from the normal logic levels. For example, connecting a bipolar transistor directly to a port pin, as in Fig. 11.4(a), will take sufficient current from the TRIS buffer to drag the pin voltage to ≈ 0.7 V, the forward conducting voltage of a typical transistor base-emitter.[4]



(a) Sourcing current *(b) Sinking current*

Fig. 11.4 Sinking and sourcing current.

The situation in Fig. 11.4(b) is similar with current flowing through the light-emitting diode (LED) into the port pin[5] and the TRIS buffer will be pulled up to ≈ 3 V assuming a conducting LED offset of 2 V.
In these situations the outcome of reading a port pin set to output is often not the state of the port bits' Data flip flop. Thus for example, `btfsc PORTB,7` in purporting to skip if bit 7 of Port B is zero may

[4] Actually somewhere between 25 mA and 35 mA (see Example 11.1).
[5] Typically around 60 mA; see Fig. 11.15.

fail to function as expected if that bit is set to output and pin RB7 is sinking or sourcing too much current.

(d) **Writing to a port configured as an input, TRIS = 1.**
In this situation the state of the Data flip flop will be altered in the proper manner. However, as the TRIS buffer is disabled, any change will not be reflected at the I/O pin until the direction of the port pin is subsequently changed to output.
This ability to set up the state of a port in a manner invisible to the outside world is important when the PIC is reset. On reset, all ports are set to input, in other words all TRIS ports are set to FF*h*. Any ports that are to control devices in the outside world should first be written to with the initial state of these devices and *only* then changed to output. For example, if four electromagnetic relays are connected to port bits RB7:4 and are to be energized after reset by logic 1 states on their input we have:

```
      org    0            ; On Reset, PortB -> all inputs
RESET movlw  b'11110000' ; Set RB7:4 to 1, RB3:0 to 0
      movwf  PORTB        ; Do it behind the scenes
      bsf    STATUS,RP0   ; Change to Bank1
      movlw  b'00001111' ; Set RB7:4 to O/P, RB3:0 to I/P
      movwf  TRISB        ; Do it exposing to outside world
      bcf    STATUS,RP0   ; Change back to Bank0
```

where we are assuming that the lower four Port B bits are to remain in their input state.

In item (c) we referred to current into (known as **sink current**) or out of (known as **source current**) the port pin. In most situation a port pin configured as an output will only be required to source or sink a few milliamps of drive current. Nevertheless, it is important to be aware of the drive capabilities of port output pins.

Generally two situations are tabulated in a device's data sheet.

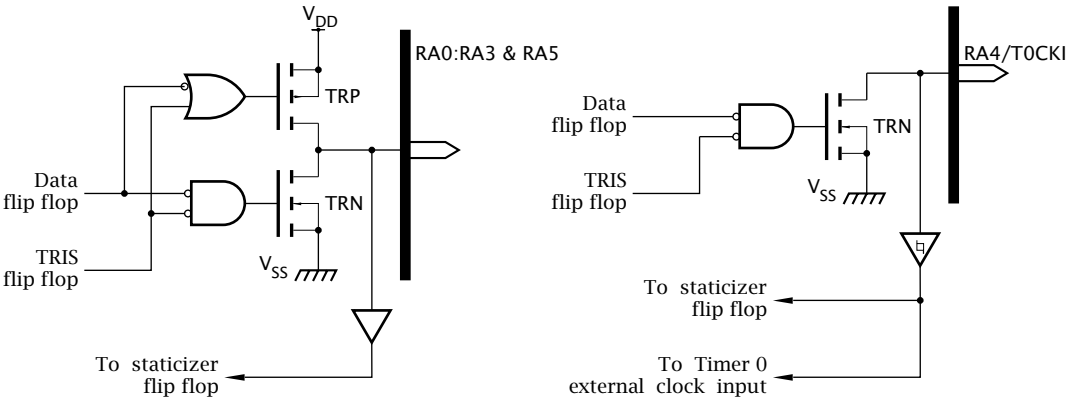
1. Sink current I_{OL} when an output is logic 0 should not exceed +8.5 mA if the low voltage V_{OL} is not to rise above 0.6 V.
2. Source current from a logic 1 output I_{OH} should not exceed -3 mA if the high voltage is not to drop more than 0.7 V below V_{DD}. The negative current denotes source; i.e. out of the device.

Larger currents may be sourced or sunk, as in Fig. 11.4, if degradation of logic levels are acceptable, subject to an absolute limitation that it must be within the range $-20 \longrightarrow +25$ mA for any single I/O pin to avoid damage. Where more than one I/O pin is involved in driving current, an overall global limit must be observed. For example, the 18-pin PIC16F83/4 limits Port A to $-50 \rightarrow +80$ mA and Port B to $-100 \rightarrow +150$ mA in total. 100 mA is the global maximum current I_{DD} into the V_{DD} pin and 150 mA the global maximum out of the V_{SS} pin. Bigger packages, such as the 40-pin PIC16C74, support larger global currents with a corresponding maximum I_{DD} of 250 mA and I_{SS} of -300 mA.

The maximum current into or out of Ports A, B and E combined is 200 mA with the same figure being specified for Ports C and D in total, subject to the global I_{DD} and I_{SS} figures and individual pin limitation.

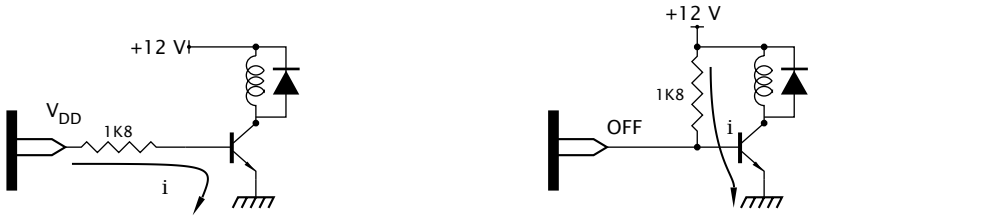
Port pins configured as inputs with normal TTL buffer inputs (Port A except RA4 and Port B) recognize an input as low where $V_{IL} \leq 0.5$ V and a high for an input $V_{IH} \geq 2$ V. Ports with Schmitt trigger input buffers (RA4 and Ports C upwards) have a V_{IL} of $0.2V_{DD}$ (1 V for $V_{DD} = 5$ V) and V_{IH} of $0.8V_{DD}$ (4 V for $V_{DD} = 5$ V).

The block diagram of Fig. 11.1 is a typical representation of a parallel port I/O bit. Specific ports may vary in ways that can affect the electrical performance in a significant manner; especially Ports A and B.



(a) Push-pull TRIS buffer

(c) Open-drain TRIS buffer



(b) Driving an electromagnetic relay

(d) Driving an electromagnetic relay

Fig. 11.5 Port A I/O pin driver structure.

Looking first at Port A, specifically RA3:0 and RA5, which are illustrated in Fig. 11.5(a), we see that the TRIS buffer is implemented using a series N-channel/P-channel field effect transistor totem pole.

- Where the TRIS bit is logic 1 the lower AND gate has a logic 0 output and the upper OR gate has a logic 1 output. In this situation, both transistors TRN and TRP are non conducting and the state of the Data flip flop is isolated from the I/O pin. In this situation the port bit is configured as an input.

- Where the TRIS bit is logic 0 then the complement state of the Data flip flop is gated through to both totem-pole transistors. With D low, TRN conducts and TRP is off giving a low pin voltage. With D high, TRP conducts and TRN is off giving a high pin voltage. In this situation the pin follows the state of the Data flip flop with current being sourced or sunk through the relatively low resistance active conducting transistors.
  As an example, consider the situation where an electromagnetic relay is to be activated from pin RA0 and requires a 200 mA activation current at 12 V. For currents and voltages of this magnitude we need external buffering. In Fig. 11.5(d) a bipolar transistor acts as an external switch. If the minimum gain of this transistor is 100 then a 1.8 kΩ resistor will give a base current of 2 mA assuming a base-emitter conduction voltage of 0.7 V and a PIC V_{OH} of at least 4.3 V.
  The output of RA4 shown in Fig. 11.5(c) is somewhat different in that only the bottom totem-pole transistor is implemented. As opposed to the 3-state structure of Fig. 11.5(a), this structure has only two states; that is active logic 0 and open-circuit. This type of output is known as **open drain** (or open-collector), see Fig. 2.3 on page 20.
- When the TRIS flip flop is logic 1, its reset state, then the AND output is low and TRN is off with the output pin high resistance. RA4 is then set to input.
- When TRIS is logic 0 the output transistor conducts when the Data flip flop is logic 0 giving an active-low output. When the Data is logic 1, TRN is off and the output floats.

An open-drain output cannot source current; either the load itself must be connected from the output pin to a positive voltage or an external pull-up resistor used as a load for the on-chip transistor. This is the case in Fig. 11.5(d) where the base current for the external transistor is derived from the 1.8 kΩ pull-up resistor when RA4 is off.

There is one further difference between RA4 and RA0:3/RA5. The distinction is in the use of a Schmitt trigger buffer to give a better noise immunity when RA4 is used as the input to Timer 0 – see Fig. 13.3 on page 365. As a consequence, logic levels into RA4 are different to other Port A (and Port B) inputs. If RA4 is to be used as the Timer 0 input, it is usually configured as an input. If configured as an output, then in this situation PORTA[4] must be set to logic 1, which will disable the open-drain transistor and prevent interaction between it and the external clock input to the Timer.

Many applications involve reading the state of arrays of switches. Rather than use the relatively more expensive single-pole double-throw (SPDT) switch arrangement of Fig. 11.6(a) to give the two logic states, most switches; for example, those in the keypad of Fig. 11.8, are single-throw (SPST) types. In these situations an external pull-up resistor is needed to

Fig. 11.6 Interfacing switches to a port line.

convert the open-circuit state to a high voltage, as shown in Fig. 11.6(b). A similar situation arises when open-drain/collector electronic devices, such as phototransistors, are to be read by a port. The value of such pull-up resistors should not be too low, as a large current will flow through the switch when closed, nor too high to reduce noise induced by electromagnetic means from external sources. A good compromise is in the range 10 – 100 kΩ.

In order to simplify the interface of such devices, Port B *inputs* have optional internal pull-up resistors. These internal resistors are called **weak pull-ups** as their typical equivalent values of around 20 kΩ is high enough not to interfere with devices being read which have 'normal' logic low and high outputs.

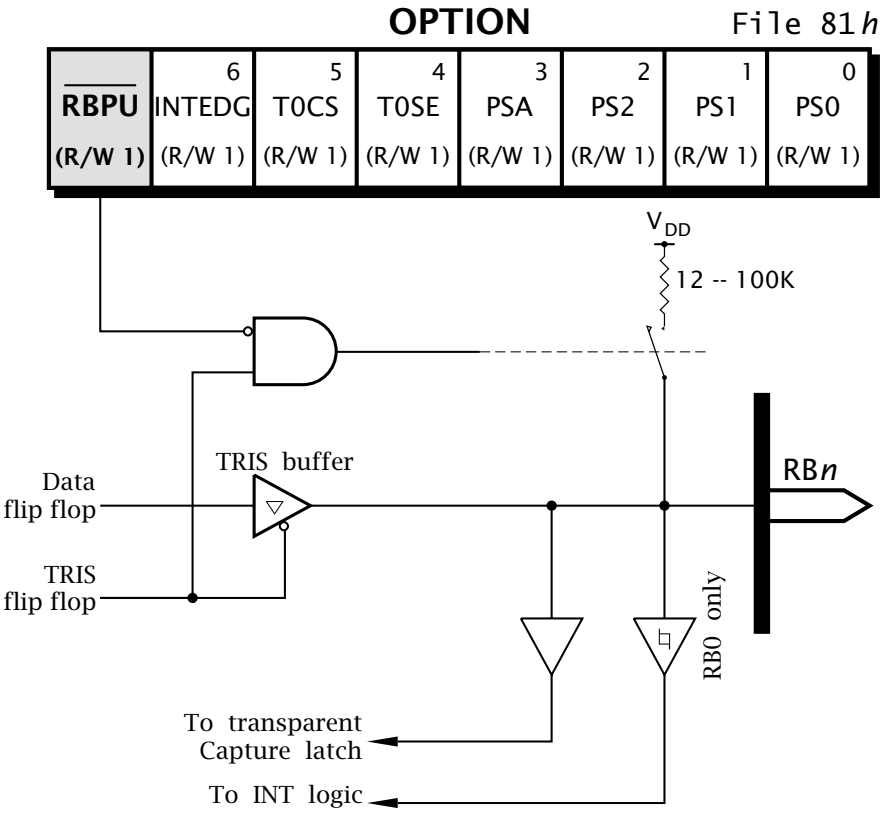
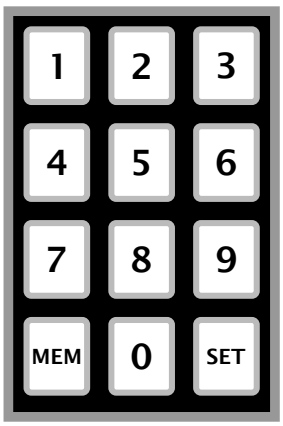


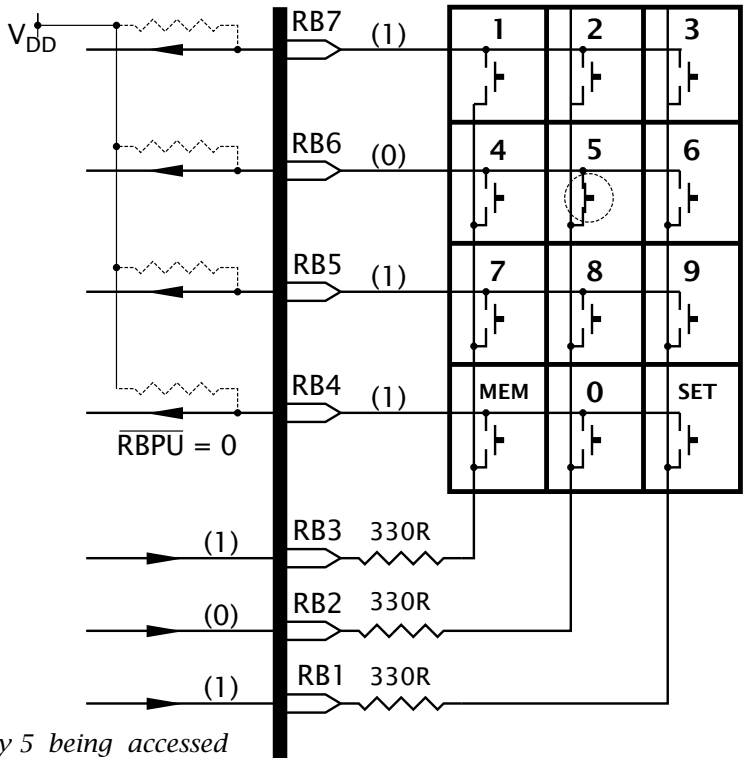
Fig. 11.7 Port B's weak pull-up option.

We see from Fig. 11.7 that the internal pull-up resistors (actually a P-channel FET) are switched in only if $\overline{\text{RBPU}}$ (Register B Pull Up, bit 7) of the Option register is low. Although all eight pull-ups are qualified by $\overline{\text{RBPU}}$ only those pins configured as inputs (TRIS[n] = 1) will have the resistor switched in. $\overline{\text{RBPU}}$ resets to 1 and so the pull-up resistors default to off.

As a typical application of weak pull-ups, consider the problem of reading a keypad, such as that illustrated in Fig. 11.8(a). In this particular example there are 12 switches and rather than use up all these scarce I/O pins it is hardware efficient to connect these switches in the form of a 4×3 matrix, as illustrated in Fig. 11.8(b). This 2-dimensional array reduces the I/O pin count to 7. Larger keypads show an even greater efficiency gain, with a 64-contact 8 × 8 keyboard needing only 16 I/O pins.



(a) A 4x3 keypad



(b) Showing key 5 being accessed

Fig. 11.8 Interfacing to a keypad.

Although there are variations on this theme, the topology shown here is typical. The four rows are read in via RB7:4 with internal pull-up resistors enabled. The three columns connected to RB1:3 can be individually selected in turn by driving the appropriate pin low, thus scanning through the matrix. The switch contacts are normally open and, because of the pull-up resistors, read as logic 1. Should a switch connected to a low column line be closed then the appropriate row line is low. This means that

once the closed key row has been detected the column:row intersection is known. The 330 Ω resistors limit the current through the switch should one of the RB7:3 pins accidentally give a low output due to erroneous software.

In order to tie these concepts together, consider a subroutine to interrogate the keypad and return either with the key pressed (or at least the first key found if more than one) or if no key then −1 (i.e. FF*h*). Before looking at the coding we can assume that somewhere in the main software Port B has been configured appropriately with the correct input and outputs assigned and that bit $\overline{\text{RBPU}}$ in the Option register has been cleared. Something like:

```
     include "p16f84.inc"
MAIN bsf    STATUS,RP0          ; Change to Bank1 where
     movlw  b'11110000'         ; TRISB & OPTION_REG lie
     movwf  TRISB               ; RB7:4 inputs, RB3:0 outputs
     bcf    OPTION_REG,NOT_RBPU ; Activate internal pull-ups
     bcf    STATUS,RP0          ; Go back to Bank0
```

The listing of Program 11.1 is based on the task list:

1. Set KEY_COUNT to one.
2. For i = 0 to 2.
   - Activate column i.
   - For j = 0 to 3.
     - Check row j.
     - IF zero THEN BREAK to step 4.
     - ELSE advance KEY_COUNT by 3.
   - Reset KEY_COUNT back to j+1.
3. Set KEY_COUNT to −1 if no key found.
4. Return KEY_COUNT.

Basically the sequence of operations is to begin with a count of one; i.e. key[1], and bring column[0] low. As each row is checked for a zero, the count kept in the Working register is advanced by 3 (lines 21, 24, 27) to reflect the three keys in each row. If no closure (that is a 0) is found, the column key tally in memory at KEY_COUNT is advanced by one (i.e. one column) and the next column tried.

There are two ways out of the loop.

- If a 0 is found during the scan, the count in W is the desired value and the subroutine immediately returns (lines 20, 23, 26, 29).
- If the column pattern shift in line 36 results in the sample 0 arriving in bit 0 then the subroutine returns FF*h* to show no key has been found.

In the real world a subroutine like this would often read in rubbish due to switch bounce and possibly noise induced in the connections between keypad and the electronics.

One way of filtering out this unpredictability is shown in the subroutine of Program 11.2. Here the state of the keypad is interrogated using

Program 11.1 Scanning the keypad.

```
; ************************************************************
; * FUNCTION: Scans 4x3 keypad & returns with a key identifier*
; * ENTRY   : None                                            *
; * EXIT    : Key in W [MEM]=10, [0]=11, [SET]=12             *
; * EXIT    : Return -1 (FFh) if no key detected              *
; * RESOURCE: KEY, PATTERN byte vars                          *
; ************************************************************

        cblock                  ; Two global variables
         KEY_COUNT:1, PATTERN:1
        endc

SCAN_IT clrf  KEY_COUNT         ; Key 1 is the first key
        incf  KEY_COUNT,f
        movlw b'11110111'       ; The initial scan pattern
        movwf PATTERN

SLOOP   movf  PATTERN,w         ; Get scan pattern from mem
        movwf PORTB             ; Set column low
        movf  KEY_COUNT,w       ; Get Key count
; Now check each row for a zero
        btfss PORTB,7           ; Check row 1
         goto GOT_IT            ; IF zero THEN found the key!
        addlw 3                 ; ELSE inc Key count by four
        btfss PORTB,6           ; Check row 2
         goto GOT_IT            ; IF zero THEN found the key!
        addlw 3                 ; ELSE inc Key count by four
        btfss PORTB,5           ; Check row 3
         goto GOT_IT            ; IF zero THEN found the key!
        addlw 3                 ; ELSE inc Key count by four
        btfss PORTB,4           ; Check row 4
         goto GOT_IT            ; IF zero THEN found the key!

; Reach here if no closed key
        movlw -1                ; Maybe no key? return -1
        incf  KEY_COUNT,f       ; Advance Key count one column
        rrf   PATTERN,f         ; Shift scan pattern once ->
        btfsc PATTERM,0         ; Check; has the 0 reached RB0?
         goto SLOOP             ; IF not DO another column

; Scan is finished here
GOT_IT  return                  ; Return with Key count in W
```

the SCAN_IT subroutine of Program 11.1. By keeping the state of the previous reading in Data memory, any change can be detected. Only if no change over 256 readings occurs will subroutine GET_IT return with the keypad state. Depending on the quality of the keypad, ambient noise

Program 11.2 Noise filtered keypad scanning.

```
; ************************************************************
; * FUNCTION: Scans 4x3 keypad and returns with a debounced  *
; * FUNCTION: key identifier                                 *
; * ENTRY   : None                                           *
; * EXIT    : Key in W [MEM]=10, [0]=11, [SET]=12            *
; * EXIT    : Return -1 (FFh) if no key detected             *
; * RESOURCE: COUNT, NEW_KEY, OLD_KEY                        *
; * RESOURCE: Subroutine SCAN_IT                             *
; ************************************************************

        cblock                  ; Three global variables
         COUNT:1, NEW_KEY:1, OLD_KEY:1
        endc

GET_IT  clrf  COUNT             ; The no-change count zeroed
GLOOP   call  SCAN_IT           ; Raw value returned in W
        movwf NEW_KEY           ; Is new value
        subwf OLD_KEY,w         ; New and old the same?
        btfsc STATUS,Z
         goto EQUAL             ; IF same go to EQUAL

; Otherwise the readings are different, so:
        movf  NEW_KEY,w         ; Make old key = new key
        movwf OLD_KEY
        goto  GET_IT            ; and start all over again

; IF readings are the same THEN
EQUAL   incfsz COUNT,f          ; Increment count; IF not
         goto  GLOOP            ; rolled around to 00 repeat

        movf   OLD_KEY,w        ; ELSE thats it!
        return
```

and processor speed, the outcome can be improved at the expense of response time by including a short delay in the loop, or by using a 2-byte stability count.

Given that RB0 doubles as the hardware INT input,[6] it is possible to interrupt the processor when any key is pressed. By bringing RB3:1 all low and ANDing all four rows, any key closure will result in the AND gate going low. This can in turn drive the RB0/INT pin and either force the processor into its interrupt service routine or at the very least set the INTF flag in the INTCON allowing the processor to poll for keypad activity.

Another approach to an interrupt-driven keypad interface is to use the Port B Change feature shown in Fig. 11.9. The top four Port B I/O pins have a second D latch in parallel but in anti-phase to the usual input Capture latch. When the CPU reads Port B the Capture latch samples the state

[6] Note from Fig. 11.7 that the internal interrupt logic is activated via a Schmitt trigger buffer as opposed to the regular Port B TTL buffer.

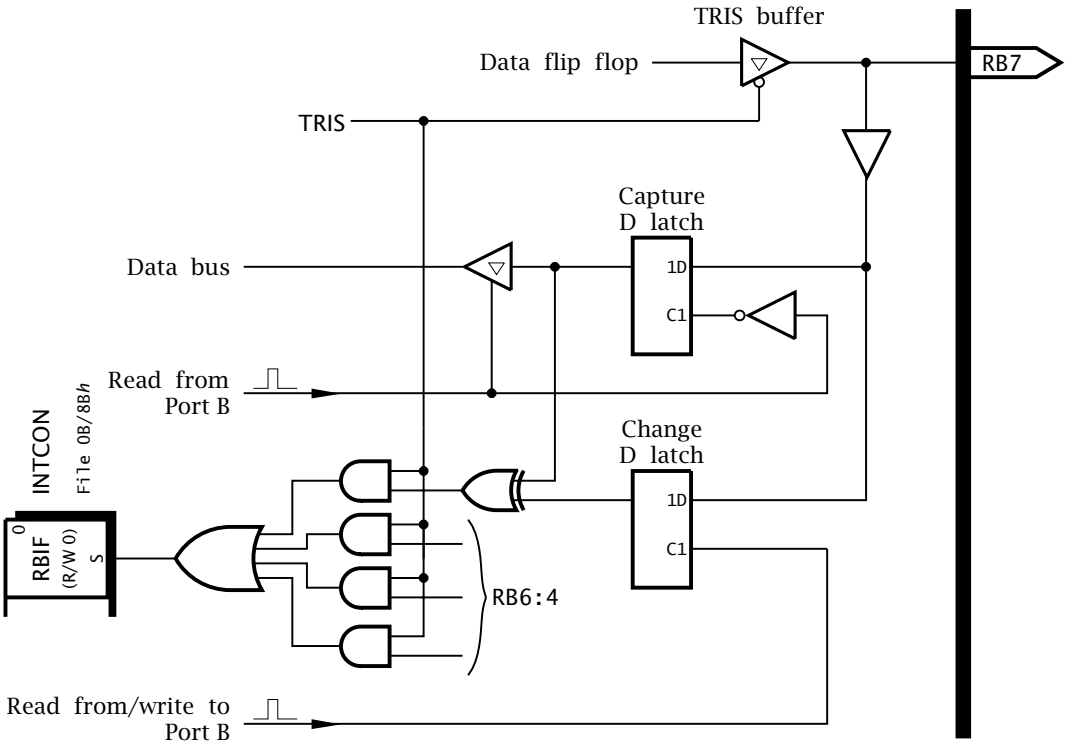


Fig. 11.9 The Port B change feature.

of the input pin in the normal way. However, at the same time the Change latch becomes transparent. When the reading action is over, the Change latch freezes and captures the pin state as it is at the time of reading. The outputs of both the Capture and Change latch are Exclusive-ORed together. As we have seen in page 14, an XOR gate detects *differences* between its two inputs. As the Capture latch is now transparent, any subsequent change at the pin input will cause the output of the associated XOR gate to go to logic 1. Each of the four Port B cells `RB7:4` has a Change feature and the four XOR gates are ORed to give a composite signal which sets the `RBIF` (Register B Interrupt Flag) in the `INTCON` of Fig. 7.4 on page 178. If the `RBIE` (Register B Interrupt Enable) bit is set then this is a convenient way of awaking a PIC slumbering in its Sleep state. If the `GIE` (General Interrupt Enable) bit is set as well, a change in the top nybble of Port B will cause an interrupt as well. Each XOR gate is ANDed with the TRIS line so that only bits that are programmed as an input can contribute to the Change signal.

Care must be taken in using this facility. For example, using the lower (non-change) part of Port B (e.g. `bclr PORTB,0`) can affect the Change facility by forcing the latches to resample. Also a change may occur at the instant the port is being read and may be missed, although later A parts (e.g. PIC16C74A versus PIC16C74) altered the sampling logic to remove

this latter problem. Neither of these foibles are factors if a keypad is used to awaken a sleeping PIC.

Once the PIC has responded to the Change interrupt the Change signal setting RBIF should be removed by reading Port B, which equates the state of the two D latches. Only then should RBIF be cleared. Failure to do this initial read will result in this interrupt flag being immediately set again.

As an example, using the keypad to awaken the PIC with the assumption that GIE is zero (no interrupt) should be implemented as:

```
    movf  PORTB,w      ; Read Port B to cancel any difference
    bcf   INTCON,RBIF  ; Clear the Change Interrupt Flag
    bsf   INTCON,RBIE  ; Enable the Change Interrupt Enable
    sleep              ; Go to sleep
; zzzzz
    call  DELAY        ; On wakening let things settle
    movwf PORTB,w      ; before cancelling any difference
    bcf   INTCON,RBIF  ; Clear the Change Interrupt Flag
    bcf   INTCON,RBIE  ; Disable Change Interrupt facility
```

Most PIC devices have relatively few I/O port lines – see Table 11.1. Even the larger footprint devices, such as the PIC16C74 with 33 peripheral pins, may not have enough parallel I/O resources for some projects, especially as several other peripheral devices may need to use the shared I/O pin budget.

As an example, consider a multi-purpose intruder alarm which can monitor up to eight zones– for example, floors in a multi-story building. Each zone can have up to eight movement sensors. A display of eight lamps back at base is to be used to indicate in which zone the intruder is located.

Based on this specification, a budget of 72 (64 input and 8 output) parallel I/O pins will be required. Rather than using one PIC device per zone reporting back to a main controller[7] it has been decided to expand the I/O capabilities of a single PIC16F84 device.

One expansion architecture is shown in Fig. 11.10. Here Port B is used to implement an external data bus which connects to the eight zone 3-state buffers and one indicator flip flop array. Each zone's set of sensors are interfaced to this local bus via an octal 3-state buffer. One of eight buffers can be enabled using a 3 to 8-line decoder addressed from Port A. For example, if RA2:RA1:RA0 were $111b$ and RA3 = 0 then Zone 7's buffers are enabled and its eight sensors can be read in at Port B.

To activate the one output lamp array, RA3 should be logic 1 and Port B set to output. Data can then be clocked into the flip flop array by pulsing RA0 low then high to give a rising edge.

The number of output ports may be expanded in this architecture to eight by using a second 3 to 8-line decoder to select the port enabled

[7] An implementation that is perfectly feasible and cost effective; see SAQ 11.1.

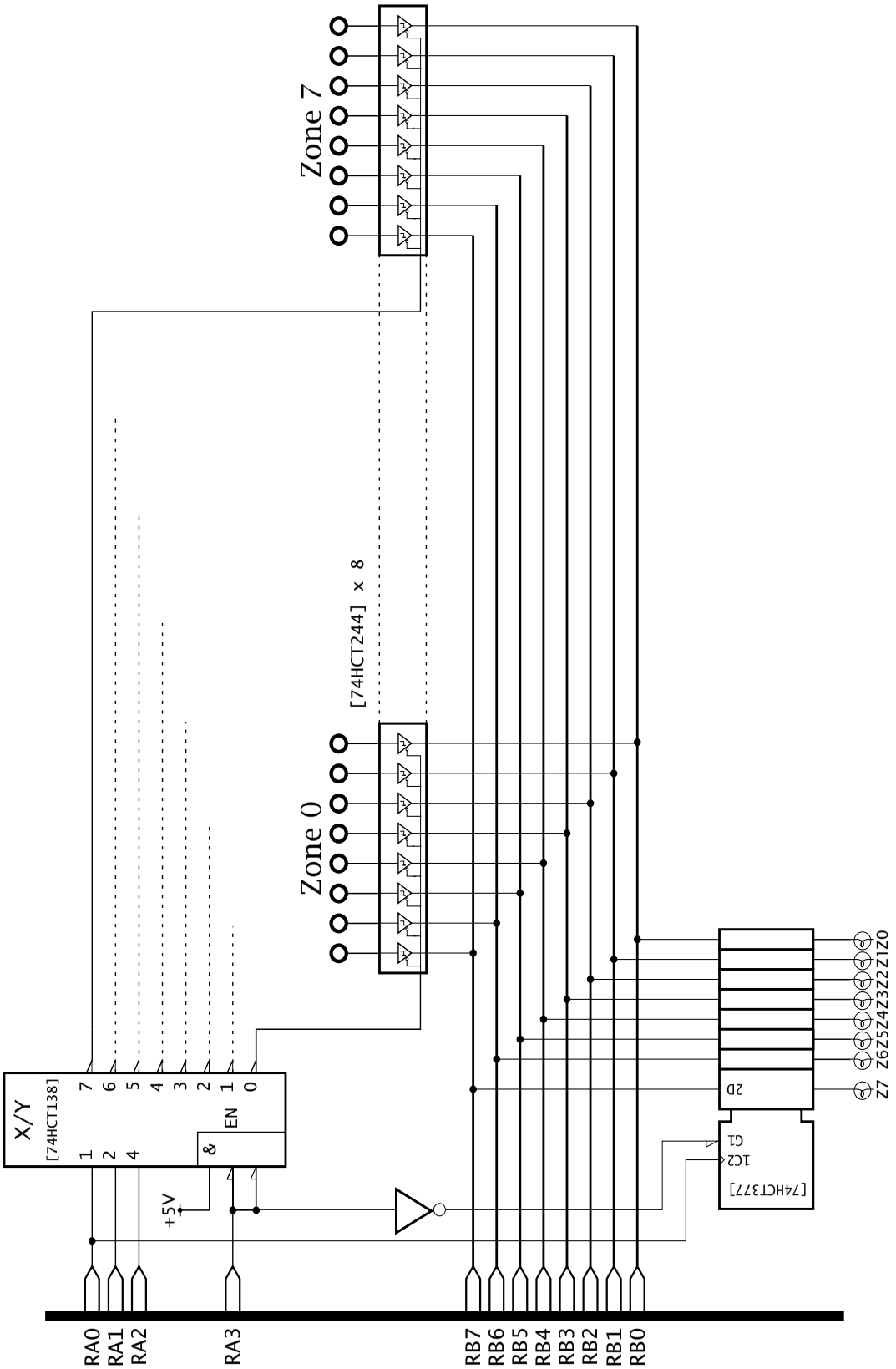


Fig. 11.10 A multi-zone intruder alarm.

when RA3 = 1. However, up to two extra output ports could be added by simply substituting RA0 by RA1 and RA2 to enable these two additional flip flop arrays. For example, one port could show which sensor(s) within the zone was active and RA4 used to sound a buzzer if any zone was active.

Program 11.3 Interacting with the intruder hardware.

```
; ************************************************************
; * FUNCTION: Reads Zone N and activates lamp N              *
; * ENTRY   : N is in file register ZONE, 00000nnn           *
; * EXIT    : Lamp N active if Zone N is non zero            *
; * EXIT    : ZONE zero and TEMP unchanged                   *
; ************************************************************
ZONE_N   bsf    STATUS,RP0    ; Change to Bank 1
         movlw  0FFh          ; Set Port B to input
         movwf  TRISB
         clrf   TRISA         ; Set Port A to output
         bcf    STATUS,RP0    ; Change to Bank 0
         movf   ZONE,w        ; Get N, used
         movwf  PORTA         ; to select Zone N's buffers
         nop                  ; Delay to allow long lines
         nop                  ; to settle
         movf   PORTB,w       ; Now read data on Port B
         btfsc  STATUS,Z      ; IF not zero THEN an intruder!
          goto  LAMP_OFF      ; otherwise all clear
; Intruder found, now activate lamp
         bsf    STATUS,RP0    ; Change to Bank 1
         clrf   TRISB         ; Port B now set to output
         bcf    STATUS,RP0    ; back to Bank 0
; Convert binary number to unary equivalent to activate lamp
         movlw  0FFh          ; All ones
         movwf  TEMP          ; into TEMP
         bcf    STATUS,C      ; Zero Carry bit
         incf   ZONE,f        ; Map zone range to 1 -- 8
Z_LOOP   rlf    TEMP,f        ; Shift pattern <--
         bsf    STATUS,C      ; Set Carry bit
         decfsz ZONE,f        ; Decrement Zone number
          goto  Z_LOOP        ; and repeat N times
; TEMP holds the unary lamp activation pattern
         movf   TEMP,w        ; Get it
LAMP_OUT bsf    PORTA,3       ; Enable output port
         movwf  PORTB         ; Lamp data
         bsf    PORTA,0       ; Clock it in by pulsing RA0
         bcf    PORTA,0
         return               ; All done
; Go here if no intruder found and turn off all lamps
LAMP_OFF bsf    STATUS,RP0    ; Change to Bank 1
         clrf   TRISB         ; Port B now set to output
         bcf    STATUS,RP0    ; back to Bank 0
         movlw  0FFh          ; All ones turns lamps off
         goto   LAMP_OUT
```

To show how this hardware interacts with the software consider the subroutine in Program 11.3 below that reads Zone *N* and if non-zero then lights lamp *N*; where *N* is an integer 0–7 in a file register called ZONE on entry. We assume that an active sensor gives logic 1 and a lamp illuminates on a logic 0.

Checking Zone *N* is simply a matter of setting Port B up as an input port and sending the Zone *N* binary pattern to Port A. The 3 to 8-line decoder is enabled whenever RA3 is low, so no processing of the Zone binary code is needed. Due to the long connection lengths, a short delay is introduced to allow data to settle. For a real system, a delay of several hundreds of milliseconds and a digital smoothing routine, such as the debounce routine of Program 11.2, would be needed for reliable data acquisition, assuming that the zone buffers were geographically distant.

Activating the eight lamps is a little more tricky. In either case, Port B must be configured as an output. The lamps are then actuated by sending the appropriate pattern to Port B, bringing RA3 high and then pulse RA0. This is implemented in Program 11.3 in routine LAMP_OUT. The lamp datum is simply all logic 1s where no intruder has been detected, that is where the sensor data has been read as all zeros.

When an intruder has been detected, then lamp *N* alone must be lit; for instance, 10111111*b* for Zone 6. To do this, the binary zone code in ZONE must be converted to the appropriate unary (one of *n*) code. For example, Zone 2 00000010*b* maps to 11111011*b*, Zone 3 00000011*b* maps to 11110111 etc.

In the program the unary code is built up in file register TEMP, which is initially set to 11111111*b*. By clearing Carry *before* entering the loop at Z_LOOP but setting it to 1 *within* the loop, a single zero can be shifted left using the Rotate Left File instruction `rlf TEMP,f`. This gives the sequence 11111111 ← 11111110 ← 11111101 ← 11111011 ... 01111111. As this shift progresses, the ZONE datum (mapped to the range 1–8 so that at least one shift is implemented) is decremented and the loop exited when this reaches zero. Thus the position of the lone 0 (the initial C = 0) represents the original zone number. This unary code is then sent out to the lamp port at LAMP_OUT to activate the one-of-n indicator.

Examples

Example 11.1

A 2N3055 NPN bipolar transistor is to be used to activate the field coils of a small stepper motor. Taking into account the minimum gain of the transistor over the range +85 → −40°C, it has been calculated that the base current must be at least 10 mA. The transistor is to be controlled from a port pin and its base-emitter voltage can be assumed to be no more than 0.7 V and V_{DD} is 5 V. What is the maximum value of the base

resistor R_B and given this value, what will be the worst-case maximum base current?

Solution

For currents of this magnitude we can assume that the pin voltage will be less than 5 V. The data sheet specifies a minimum voltage of 4.3 V (a drop of 0.7 V) for a I_{OH} of −3 mA but for currents greater than this we must resort to graphical techniques.

Figure 11.11 shows the graphical relationship of output source current I_{OH} for a high output voltage state V_{OH}. The grey area is bounded by the minimum situation, which is at +85°C and maximum condition at −40°C.

This voltage V_{OH} is also a function of the transistor input base resistor circuit according to the equation $V_{OH} = 0.7 + I_{OH} \times R_B$. This straight line relationship (called a **load line**) is shown on the graph from (0,0.7) drawn to intersect the minimum locus at a current of −10 mA. This crossover is the only point that satisfies both current-voltage relationships. The

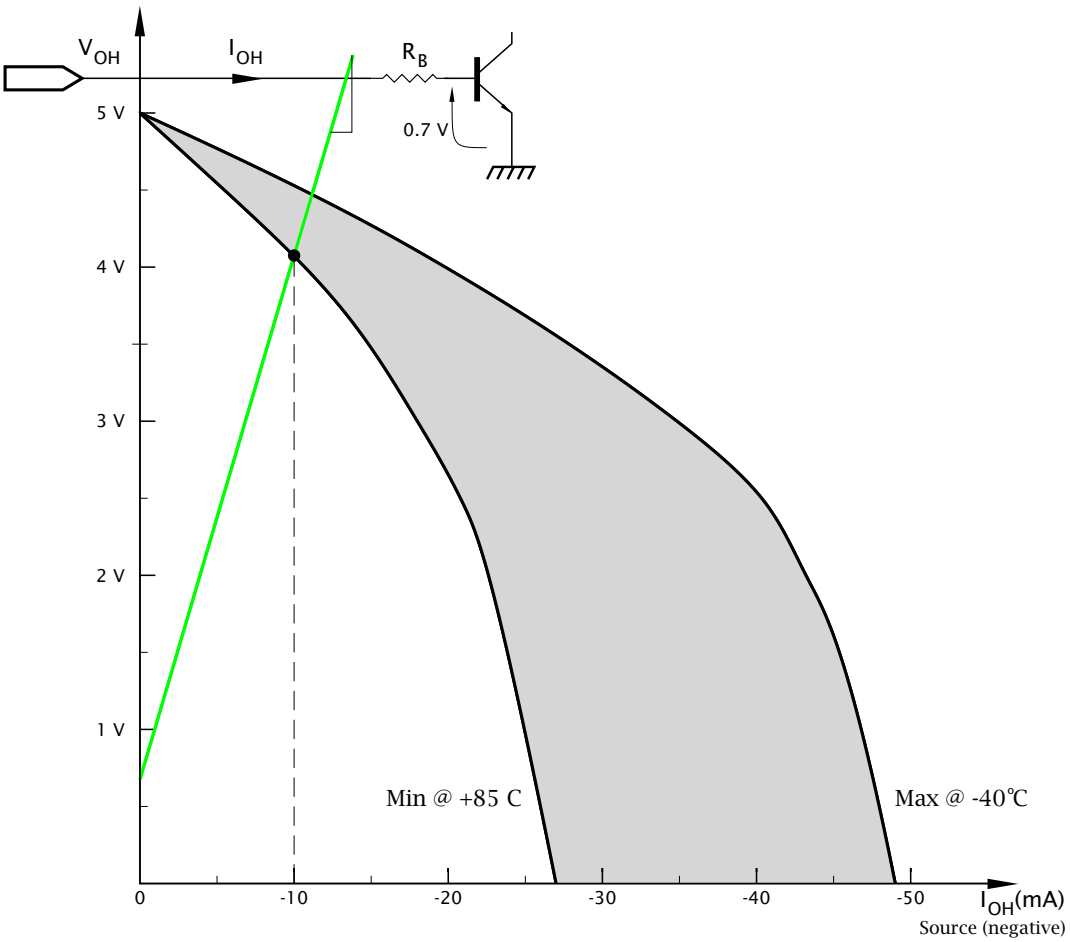


Fig. 11.11 Source current against voltage.

slope of the load line $\frac{\Delta V}{\Delta I}$ is the resistance in kΩ (as current is in mA) and measures 280Ω.

Extending the load line onwards gives the maximum current as the X co-ordinate of the intersection with the Maximum locus, which is approximately 11.5 mA; not much different. If the current requirement had been larger, then the minimum/maximum currents diverge showing a significant temperature sensitivity. For example, a 20 mA minimum base current requires a base resistor of $\approx$ 120Ω (assuming a base voltage of 0.8 V) and the maximum base current would be 28 mA.

Example 11.2

An 18-pin mid-range PIC is to be used as a digital comparator where a parallel-input 8-bit word P is to be compared to a byte datum located in a file register named TRIP. Outputs are to indicate Lower-Than, Equivalent and Higher-Than. The comparator is to have an hysteresis of ± 1 bit. That is, if previous comparisons showed $N <$ TRIP then the trigger level is increased to TRIP + 1 for equality. Similarly, on a downward trajectory the equality level is to be decreased to TRIP − 1.

Datum P is to be input via Port B set up as input and the lower three Port A pins give the active-high comparator outputs <, ==, > at RA2, RA1, RA0 respectively.

Solution

The task list for such a specification is:

1. Subtract P from LEVEL.
2. IF Equal (EQ when Z=1) THEN == output active.
3. ELSE IF P Higher than LEVEL (HI when C=0, Borrow) THEN > output active AND LEVEL = TRIP − 1.
4. ELSE IF P Lower than LEVEL (LO when C=1, No Borrow) THEN < output active AND LEVEL = TRIP + 1.

The subroutine given in Program 11.4 assumes that the main program has set up the port directions accordingly and the fixed value is in TRIP. Initially LEVEL would have been set to the same value as TRIP but would subsequently vary by ± 1 as per the specification – the hysteresis band.

Software solutions to traditional hardware functions, such as comparison, have the advantage of greater flexibility, albeit at the price of a lower data throughput. Using low-cost 'computing engines', such as the PIC, means that relatively simple functions traditionally implemented by dedicated hardware can be replaced by embedded processors.

In this instance, flexibility could be replacing the fixed trip level by a variable datum input via, say, Port C – requiring a larger footprint device; eg. PIC16C74 (see SAQ 11.7). Example 12.1 on page 349 shows how an external datum can be serially acquired externally. Alternatively, an analog signal could represent one or both of the levels in devices with integral A/D converters – see Example 14.7 on page 428. In all these situations

Program 11.4 A digital comparator with hysteresis.

```
COMP      movf   PORTB,w      ; Get input P
          subwf  LEVEL,w      ; LEVEL - P
          btfss  STATUS,Z     ; Skip if equality
           goto  CONTINUE     ; ELSE IF not THEN try alternative

; This code for equality
          movlw  b'11111010'  ; Make == output logic 1
          movwf  PORTA        ; Other outputs logic 0
          goto   COMP_END     ; and exit

CONTINUE  btfsc  STATUS,C     ; Skip if borrow (P higher than)
           goto  LO           ; ELSE P < LEVEL

; This code if P > LEVEL
HI        movlw  b'11111001'  ; Set > output RA0 to logic 1
          movwf  PORTA        ; Rest to 0
          decf   TRIP,w       ; Copy TRIP-1 to w
          movwf  LEVEL        ; The new comparator level
          goto   COMP_END     ; and exit

; This code when P < LEVEL
LO        movlw  b'11111100'  ; Set < output RA2 to low
          movwf  PORTA        ; Rest to 0
          incf   TRIP,w       ; Copy TRIP+1 to w
          movwf  LEVEL        ; The new comparator level

COMP_END return
```

the hysteresis may advantageously be made a fraction of the trip voltage, eg. $\pm\frac{1}{32}$, rather than a fixed ± 1 bit.

Example 11.3

The principle of a stepper motor is shown in Fig. 11.12. In essence there are four coils, labelled A, B, C, D, which may be selectively energized either singly or in pairs, to generate a magnetic field in one of eight directions in divisions of $45°$.[8] Thus Coil A alone gives a northerly field, A & B together give a north-easterly direction, B alone is east, etc. The rotor follows the field as it changes direction provided that inertial considerations allow it to keep up during acceleration and de-acceleration.

Solution

Our first step is to devise a table showing energization patterns for the eight possible field directions, as shown in Table 11.2.

[8] A real stepper motor repeats the coil set several times around the peripheral motor stator giving a finer mechanical step resolution. Thus, if there are four sets of stator coils the $45°$ electrical step translates to $11.25°$ mechanical.

Program 11.5 Driving a stepper motor.

```
    #define  FREQ  d'40' ; Programmer gives value in 100k steps
    org      050h        ; Code begins at 050h
; ************************************************************
; * FUNCTION: Advances stepper motor 1 -- 256 steps          *
; * ENTRY   : Step number in STEP                            *
; * ENTRY   : Current field position in POSITION             *
; * EXIT    : POSITION updated, STEP = -1, W destroyed       *
; * RESOURCE: Subroutine PATTERN, DELAY_10MS                 *
; ************************************************************
MOTOR   incf   POSITION,w  ; Advance field direction
        andlw  b'0111'     ; Module-8
        movwf  POSITION    ; updated
        call   PATTERN     ; Get the energization pattern
        movwf  PORTA       ; Send to stepper motor
        call   DELAY_10MS  ; Hold off 10ms
        decfsz STEP,f      ; Decrement step count
         goto  MOTOR       ; until zero
        return
; ************************************************************
; * FUNCTION: Maps an integer 0 -- 7 to field pattern        *
; * ENTRY   : Modulo-8 integer in W                          *
; * EXIT    : Stepper energization pattern in W              *
; ************************************************************
PATTERN addwf  PCL,f       ; Increment Program Counter
        retlw  b'1000'     ; North
        retlw  b'1100'     ; North east
        retlw  b'0100'     ; East
        retlw  b'0110'     ; South east
        retlw  b'0010'     ; South
        retlw  b'0011'     ; South west
        retlw  b'0001'     ; West
        retlw  b'1001'     ; North west
; ************************************************************
; * FUNCTION: Delays 10 ms delay independent of clock freq   *
; * ENTRY   : FREQ is xtal frequency in multiples of 100kHz  *
; * EXIT    : 10ms delay; DELAY zero, W destroyed            *
; ************************************************************
DELAY_10MS
       movlw   FREQ        ; The programmer's statement
       movwf   TEMP        ; Gives the PIC frequency
; Delay loop 10ms at f = 100kHz xtal  (1 cycle = 40us)
DLOOP1 movlw   d'62'       ; Loop count
       movwf   DELAY
DLOOP2 decf    DELAY,f     ; 62 * 40us
       btfss   STATUS,Z    ; 62 * 40us
        goto   DLOOP2      ; 62 * 80us
       decfsz  TEMP,f      ; Decrement frequency parameter
        goto   DLOOP1      ; and repeat until zero
       return
```

Table 11.2: Energization pattern for the eight field directions.

Position	A	B	C	D	Bearing
0	1	0	0	0	↑
1	1	1	0	0	↗
2	0	1	0	0	→
3	0	1	1	0	↘
4	0	0	1	0	↓
5	0	0	1	1	↙
6	0	0	0	1	←
7	1	0	0	1	↖

The coding shown in Program 11.5 comprises three subroutines.

MOTOR
The main subroutine simply modulo-8 increments the position vector by post-ANDing with $00000111b$ to give a wrap around from 7 to 0. This vector is then converted to the appropriate energizing pattern and sent out to the motor after a nominal 10 ms delay. The process is repeated until the decrementing STEP datum reaches zero – if initially zero then 256 steps will be actioned.

PATTERN
Returns one of eight energization patterns corresponding to the field vector as listed in Table 11.2. The mechanism of this look-up table coding has been described in Program 6.4 on page 149. As this suite of subroutines originates at $050h$, the 8-bit addition to the Program Counter will not result in roll over across boundaries.

DELAY_10MS
This subroutine gives a nominal 10 ms delay independent of the processor crystal frequency, as defined by the programmer in the program

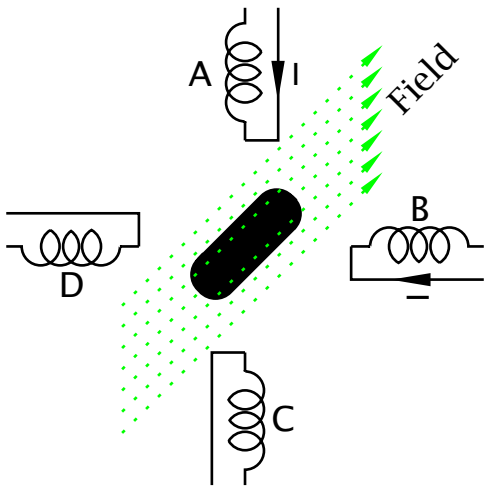


Fig. 11.12 The stepper motor.

header as a number FREQ in steps of multiples of 100 kHz. Thus for a 8 MHz crystal, giving a 2 MHz machine cycle, FREQ should be defined as 80 using the #define directive, before the program is assembled.

The core of the subroutine is a loop needing a nominal 10 ms execution time at a crystal frequency of 100 kHz – 40 μs machine cycle. This loop is transversed FREQ times. Thus our 8 MHz example will have a loop execution of $\frac{10}{80}$ ms but will be executed 80 times to give our required 10 ms delay.

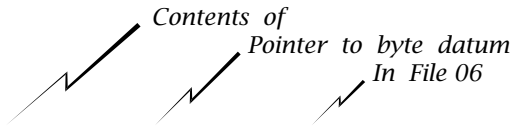
Example 11.4

Redo the keypad driver of Program 11.1 but coded in **C**.

Solution

Software structures of this nature, that is interacting with peripheral devices, are classified as **device drivers**. Device drivers or handlers have to be able to get at individual register bits in an efficient and sometimes real-time manner. Thus, even in a software system coded in a high-level language, the device drivers are traditionally written at assembly level. However, it is possible to code most device drivers in **C**, especially where response time is not critical.

The essence of the use of **C** in interacting with the various peripheral interface devices lies in its ability to operate at the colloquially called **bit twiddling** (or bit banging) level. To do this the programmer must be able to access fixed addresses in the Data store and to monitor or change individual bits within a datum. We have already seen on page 242 how to directly access a known memory location. For example, the definition:




```
#define  PORT_B  *(unsigned int *)0x06
```

defines the name PORT_B as synonymous with the contents of File 6, that is Port B.

Any bit or bits in, say, Port B can be monitored by using the & AND operation ; for example,

```
if((PORT_B & 0x80) == 0)  {do this;} /* Check bit7 for 0 */
if((PORT_B & 0x02) != 0)  {do that;} /* Check bit1 for 1 */
```

executes the statement {do this;} if bit 7 is zero and {do that;} if bit 1 of Port B is a one.

Most microcontrollers have native instructions to bit twiddle single bits *directly* in memory in a single execution cycle. Where only one bit

is involved this is more efficient than the use of AND and OR operations, and because of this **C** compilers designed to be used for such hardware targets usually have (non standard) operators designed to make use of this feature.

On page 244 the CCS compiler `#bit` declaration was used to define individual bits that could subsequently be tested by code. Using this technique our example becomes:

```
#bit  B7 = 06.7     /* PortB, bit 7                          */
#bit  B1 = 06.1     /* PortB, bit 1                          */
.....................
if(!B7)  {do this;} /* Do this if Bit7 is false (i.e. 0) */
if(B1)   {do that;} /* Do that is Bit1 is true  (i.e. 1) */
```

The CCS compiler comes with a **header file** for each processor which includes a bit description of all that device's Special-Purpose Register set and I/O pins. Our examples assume that we have included the file `16f84.h`. Thus:

```
if(!input(PIN_B7)) {do this;} /* Do this if bit RB7 == 0 */
if(input(PIN_B1))  {do that;} /* Do that if bit RB1 == 1 */
```

There is also a complementary `output` function; for instance:

```
output_pin(PIN_B1,0);          /* Make RB1 low             */
output_pin(PIN_B2,1);          /* Make RB2 high            */
output_pin(PIN_B3,1);          /* Make RB3 high            */
```

The CCS compiler adopts the policy that the inner workings of the various peripheral devices should be as invisible to the programmer as possible. To this end the compiler comes with a rich set of internal functions to set up and use the interface features appropriate to the target device specified by the header file.

As an example of this philosophy, `set_tris_b(0xF0);` is an alternative to the approach adopted on page 274. Similarly the internal function `port_b_pullups(TRUE);` is an alternative to setting the $\overline{\text{RBPU}}$ bit in the Option register.

The CCS compiler handles parallel I/O in several different ways. The `#use fast_io(b)` statement below leaves it up to the programmer to explicitly set up the appropriate TRIS registers. Other alternatives allow the programmer to ignore such minutia, but then the compiler will set up the port configuration *each* time it is used, even if that configuration remains unchanged from the last usage.

Program 11.6 assumes the following code as part of the main routine:

```
#include  <16f84.h>
#use      fast_io(b)
#define PORT_B  *(unsigned int *)0x06
unsigned int scan_it(void);
unsigned int get_it(void);
int main()
{
int ....;             /* Main's variable declarations  */
set_tris_b(0xF0);     /* RB7:4 outputs, RB3:0 inputs   */
port_b_pullups(TRUE); /* PortB pullups active          */
......
......
}
```

Program 11.6 Coding the keypad device driver in **C**.

```
unsigned int get_it(void)
{
unsigned int count, old_key, new_key;
count = 0;
while(count<255)
    {
    new_key = scan_it();
    if(new_key == old_key)
        { count++;}
    else
        {
        old_key = new_key;
        count = 0;
        }
    }
return (old_key);
}

/*************************************************************/

unsigned int scan_it(void)
{
unsigned int key, pattern;
key=1; pattern = 0xF7;         /* Initial pattern 11110111b   */
while(key<13)
    {
     PORT_B = pattern;
     if(!input(PIN_B7)) {break;}
     if(!input(PIN_B6)) {key+=3; break;}
     if(!input(PIN_B5)) {key+=6; break;}
     if(!input(PIN_B4)) {key+=9; break;}
     pattern = pattern >>1;
     key++;
    }
if(key==13) {key = 0xff;}
return key;
}
```

The code in Program 11.6 shadows that of Program 11.1 in that two functions are used; `scan_it()` scans the keypad once and returns with a value 1 – 12 or FF*h* if no key is pressed. Function `get_it()` repetitively calls `scan_it()` until 255 identical values are returned, and this value is the final outcome.

`scan_it()` initializes the column count `key` to 1 and the column scan pattern to 11110111*b*. This test vector is sent to PortB and each row is tested in turn adding 3, 6 or 9 to the value of `key` if a zero is found and the `while` loop is exited (`break`). If after the four rows have been tested no outcome has been detected, the column scan pattern is shifted right and `key` is incremented. The process is continued until either a `0` is found or `count` reaches 13. In the latter case a `key` value of FF*h* (−1) is returned.

`get_it()` keeps a number of tries `count` tally, last reading `old_key` and current reading `new_key` variables. `count` is incremented after calling `scan_it()` if the current reading is the same as the last reading. If not, `count` is reset and `old_key` is updated. The `while` loop exits if `count` reaches 255 (the maximum value of an `int` variable), indicating that the last 255 readings are the same.

Example 11.5

Despite the increasing use of liquid-crystal alphanumeric readouts, discrete 7-segment LED displays are commonly used to show up to six numerical digits. Such readouts are particularly effective in low ambient light situations and where large displays are needed.

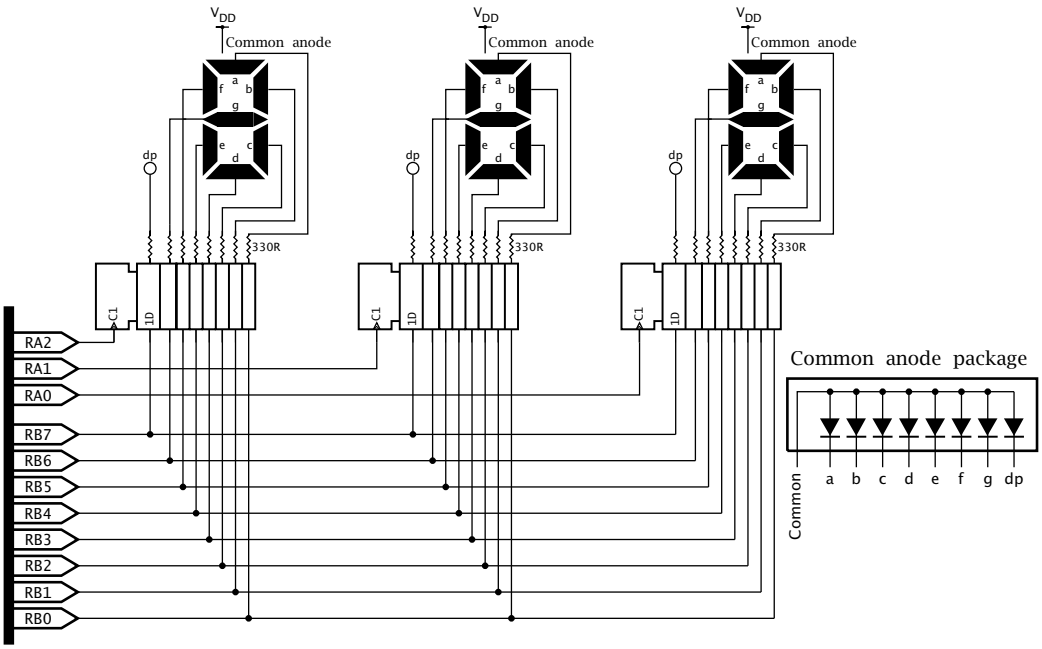


Fig. 11.13 Using port expansion to drive three 7-segment displays.

Assuming each display requires eight lines (seven segments plus decimal point) then a budget of $8 \times n$ parallel lines are required for an n-digit display. The straightforward solution to this problem is shown in Fig. 11.13, where a 3-digit display is driven from three parallel registers on a local bus, in the manner of Fig. 11.10. The principle can be extended to six or more digits using the appropriate number of registers.

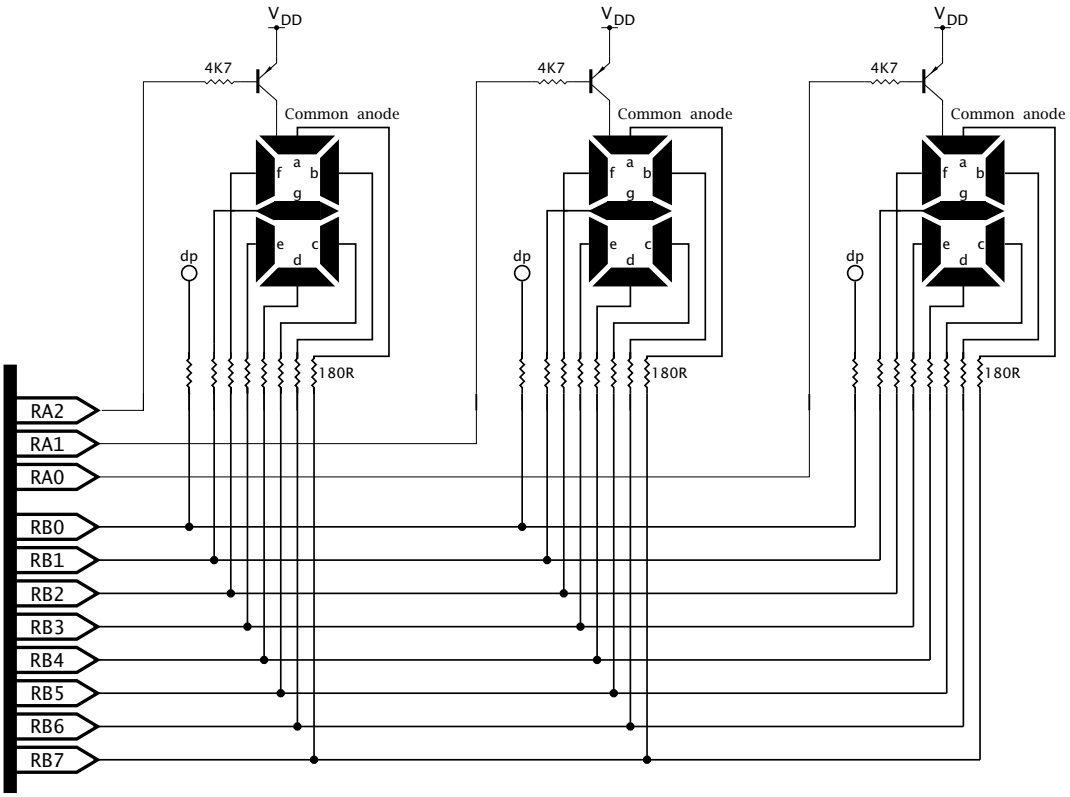


Fig. 11.14 Scanning a 3-digit 7-segment array.

The displays shown in the diagram are common anode and the appropriate LED is illuminated when the register output is low, with the sink current limited by the series resistance. In practice most logic circuitry can sink more current into a low output as compared to sourcing current from high and because of this common cathode displays are less common. In some larger displays , eg. 5 cm (2"), several LEDs may be paralleled or in series in each segment. In this situation larger anode voltages and/or currents may be needed and suitable drivers required to buffer the register outputs.

An alternative approach, shown in Fig. 11.14, is frequently used with LED-based displays. Instead of using a register for each digit, all readouts are connected in *parallel* to the one PIC port. Each readout is en-

abled in turn for a short time with the appropriate data from the output port. Provided that the scan rate is greater than 50 per second (preferably greater than 100) the brain's persistence of vision will trick the onlooker into visualizing the display as flicker free.[9] Of course the current flowing through the segment must be increased to compensate for the mark:space ratio but LEDs are more efficient at larger current pulses and the reduction of series resistance need not be pro-rata.

Discuss the pros and cons of these two arrangements with particular reference to the tradeoff of software and hardware. Illustrate your answer by displaying the decimal equivalent of the binary byte in File 20*h*. For example if the contents of BINARY were FF*h* then the display should be 255.

Solution

From the software perspective two main functions can be identified. Firstly the binary code in File 20*h* has to be decomposed into three BCD digits; HUNDREDS, TENS and UNITS. Once this is done then each BCD digit ranging from 0–9 must be converted to 7-segment code to illuminate the relevant segments to form the appropriate characters. We already have a subroutine to implement the former in Program 6.10 on page 161 and the latter in Program 6.4 on page 149. Based on this code in situ, we have as a task list for software to interact with the hardware of Fig. 11.13:

1. Convert the binary byte into BCD.
2. DO
   (a) Copy contents of HUNDREDS into W and convert to 7-seg.
   (b) Copy 7-segment code to Port B.
   (c) Pulse __/‾__ RA2.
3. DO
   (a) Copy contents of TENS into W and convert to 7-seg.
   (b) Copy 7-segment code to Port B.
   (c) Pulse __/‾__ RA1.
4. DO
   (a) Copy contents of UNITS into W and convert to 7-seg.
   (b) Copy 7-segment code to Port B.
   (c) Pulse __/‾__ RA0.

The coding implementing this task list is shown in Program 11.7.

The interaction of the software to the hardware of Fig. 11.14 is not so straightforward as there are no registers to dump the data and run! Instead, data has to be continuously sent out in sequence with the appropriate display being enabled. If we use a scan rate of 100 updates each second then this data should be held for 10 ms before moving on. Thus we have as our new task list:

1. Convert the binary byte into BCD.
2. DO forever:

[9] Of course this is how the brain interprets a series of 24 still frames per minute in a movie, each shown twice, as a moving image.

Program 11.7 Displaying the decimal equivalent of a binary byte.

```
; Task 1
DISPLAY movf  BINARY,w     ; Get binary byte
        call  BIN_2_BCD    ; Convert to 3-digit BCD
; Task 2
        movf  HUNDREDS,w   ; Get Hundreds nybble
        call  SVN_SEG      ; Convert to 7-segment code
        movwf PORTB        ; Send out to PortB
        bsf   PORTA,2      ; Clock into register
        bcf   PORTA,2

; Task 3
        movf  TENS,w       ; Get Tens nybble
        call  SVN_SEG      ; Convert to 7-segment code
        movwf PORTB        ; Send out to PortB
        bsf   PORTA,1      ; Clock into register
        bcf   PORTA,1

; Task 4
        movf  UNITS,w      ; Get Units nybble
        call  SVN_SEG      ; Convert to 7-segment code
        movwf PORTB        ; Send out to PortB
        bsf   PORTA,0      ; Clock into register
        bcf   PORTA,0
```

(a)
- Copy contents of HUNDREDS into W and convert to 7-segment code.
- Copy 7-segment code to Port B.
- Bring RA2 low ‾_.
- Delay 10ms.
- Bring RA2 high _/‾.

(b)
- Copy contents of TENS into W and convert to 7-segment code.
- Copy 7-segment code to Port B.
- Bring RA1 low ‾_.
- Delay 10ms.
- Bring RA1 high _/‾.

(c)
- Copy contents of UNITS into W and convert to 7-segment code.
- Copy 7-segment code to Port B.
- Bring RA0 low ‾_.
- Delay 10ms.
- Bring RA0 high _/‾.

The coding in Program 11.8 makes use of the 10 ms delay subroutine illustrated in Program 11.5 to regulate the scanning rate. Apart from the length of the enabling pulse the core of the program is identical to our previous situation. However, the code must run continually to give the impression of a constant display. This illustrates the trade off between hardware and software. Reducing the hardware has lead to greater load-

Program 11.8 Displaying a 3-digit decimal number on a scanning readout.

```
; Task 1
DISPLAY movf  BINARY,w     ; Get binary byte
        call  BIN_2_BCD    ; Convert to 3-digit BCD
; Task 2(a)
LOOP    movf  HUNDREDS,w   ; Get Hundreds nybble
        call  SVN_SEG      ; Convert to 7-segment code
        movwf PORTB        ; Send out to PortB
        bcf   PORTA,2      ; Enable Hundreds display
        call  DELAY_10MS   ; for 10ms
        bsf   PORTA,2      ; and turn off

; Task 2(b)
        movf  TENS,w       ; Get Tens nybble
        call  SVN_SEG      ; Convert to 7-segment code
        movwf PORTB        ; Send out to PortB
        bcf   PORTA,1      ; Enable Tens display
        call  DELAY_10MS   ; for 10ms
        bsf   PORTA,1      ; and turn off

; Task 2(c)
        movf  UNITS,w      ; Get Units nybble
        call  SVN_SEG      ; Convert to 7-segment code
        movwf PORTB        ; Send out to PortB
        bcf   PORTA,0      ; Enable Units display
        call  DELAY_10MS   ; for 10ms
        bsf   PORTA,0      ; and turn off

        goto  LOOP         ; DO forever
```

ing on the software. Indeed, as illustrated here the entire existence of the PIC will be to service the display! However, in practice the situation can be redeemed somewhat by interrupting the PIC at 10 ms intervals to avoid the need for time-wasting delay routines. The listing on page 380 shows how this can be done, but of course the Timer cannot be used for anything else. Alternatively an external 100 Hz oscillator can be used in its place, but some of the hardware advantages are then lost.

Another issue that can occur with scanning, is noise introduced by switching on relatively large currents on a continual basis. This can be a particular problem where analog circuitry is adjacent. Good power-supply decoupling can reduce this problem to some extent.

Self-assessment questions

11.1 One problem with the intruder alarm configuration of Fig. 11.10 is the need to cable the Zone ports with eight conductors plus one per

zone. An alternative approach would be to replace each zone's 3-state buffer by a PIC. Each PIC would drive a 4-wire common bus back to the main base PIC. One wire can be used as a shared handshake line to signal the base that an intruder has been sensed at the zone indicated on the three data wires.

Show how a PIC16F84 could be configured as a Zone PIC paying particular attention to the usage of the single handshake line shared with all zones.

Would it be possible to reduce the number of wires to three? How could a local display be added to show which sensor has been set off?

11.2 A certain PIC running at 20 MHz has its Port C connected to LEDs tied high through a 1 kΩ resistor and with a 300 pF capacitance to ground. All LEDs are off and the programmer attempts to turn on LED 7 and LED 0 as follows:

```
    bcf    PORTC,7  ; Turn on LED7
    bcf    PORTC,0  ; Turn on LED0
```

However, only LED 0 actually turns on. What is happening?

11.3 A certain system needs to be able to both activate eight LEDs and to be able to read the state of up to eight normally-open (N.O.) push switches. It has been proposed that a single Port B might be able to combine these functions – the former when set to output, the latter when set to input. Can you devise a suitable circuit?

11.4 A PIC-based reaction meter is to be designed to act as a crude blood-alcohol level indicator. The principle of the device is that a buzzer is sounded for 100 ms when the unseen tester closes his or her switch. The subject is to respond to the sound by immediately pressing his/her switch. An 8-LED barograph display is to indicate the passage of time by progressively illuminating an additional LED every 50 ms. The number of lit LEDs at the conclusion of the test is the reaction time in 50 ms steps.

Show how a PIC16F84 could be configured in hardware and software to read both switches and activate the eight LEDs and one buzzer.

11.5 The variation of logic 0 output voltage V_{OL} against sink current I_{OL} for the two extremes of the commercial temperature range is shown in Fig. 11.15. Using this graphical relationship determine the maximum value of series resistor to ensure a current of no less than 20 mA will

flow through an LED connected to +5 V, as shown in the diagram, for any temperature. With this value what will be the current be at −40°C? You can assume that the conducting voltage across the LED is a constant 2 V.

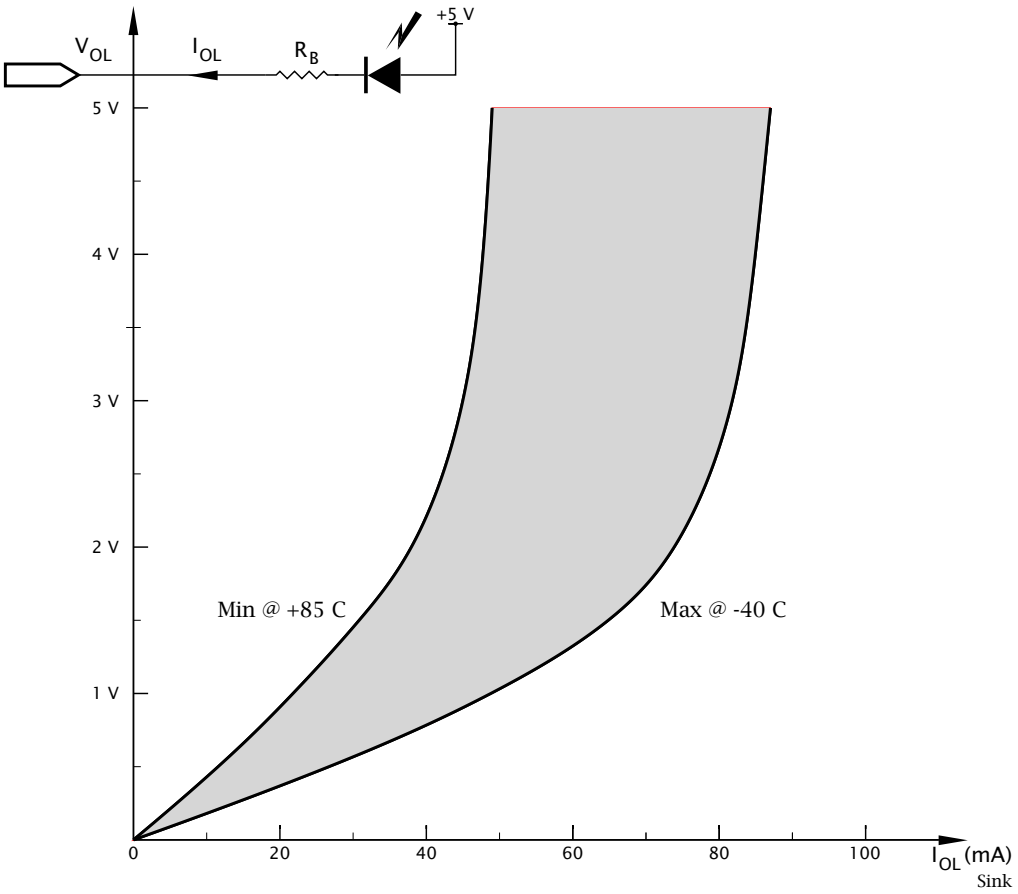


Fig. 11.15 Low-level output voltage against sink current.

11.6 Repeat the exercise of SAQ 11.4 but coding in **C**.

11.7 Extend the digital comparator of Example 11.2 to compare two *external* digital bytes presented to a 28-pin footprint PIC, with byte *P* being input at Port B and *Q* at Port C.

11.8 In a low-power wireless data logging system placing the PIC in its sleep mode will not affect the current consumption of the radio transmitter. It is proposed to use a port pin to supply current to the transmitter and in way this auxillary circuitry can be switched on and off as necessary. Discuss.

CHAPTER 12

One Bit at a Time

Parallel data transmission is fast, with a minimum of software overhead. However, there are circumstances where its use is inappropriate; either because of the additional hardware cost (see Fig. 11.10 on page 287) or more commonly where the receivers are geographically distant, with the concomitant cost or non availability of multiple communication channels and their necessary interface hardware. In such situations data can be sent one bit at a time and assembled by the remote device into the original data bytes. In this manner a comparison can be made with the parallel port on a PC, commonly used for local peripherals, such as a printer, and the serial port frequently used with a modem to link into the internet via a single telephone line.

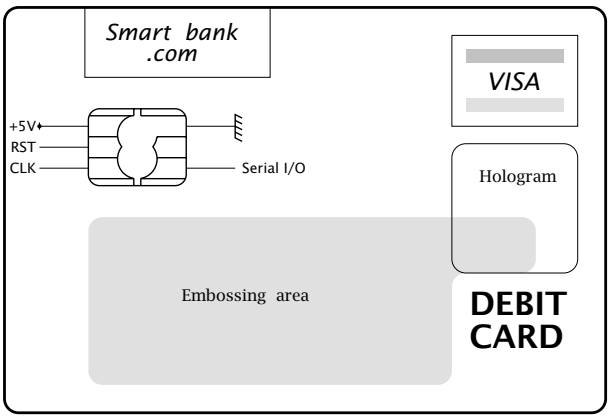


Fig. 12.1 The smart card.

As an example, consider the smart cards in your wallet. Each card will have an embedded microcontroller, typically 8-bit, giving it its intelligence. Cost constraints are severe to give a manufacturing price of under $1, and a large component of this is accounted by the non corrosive gold-plated contacts via which the microcontroller is powered and clocked when in contact with the card reader. In order to keep the mechanical precision of the reader low and hence reliability high, the number of contacts must be minimized and pad size maximized.

The standard arrangement shown in Fig. 12.1 uses contacts to provide the two power nodes, Reset, Clock and *one* line to allow data to be shifted in or out *one bit at a time*. Although this is relatively slow, in comparison to the human-mechanical constraints speed is not an issue. Furthermore, contact between the reader/automatic teller and the central computer, perhaps several thousands of miles/kilometers away, will typically be via a single channel telephone or ISDN line.

In this chapter we will examine a range of techniques used to serially transmit data, both using bespoke shift register circuits and industrial devices using standard communication protocols. After reading this chapter you will:

- Understand the need for serial transmission.
- Be able to design serial ports and associated software routines to communicate with standard parallel peripheral devices.
- Be capable of interfacing serial peripheral devices using both the SPI and I^2C protocols.
- Appreciate the need for asynchronous serial communication and be able to write software drivers conforming to this protocol.
- Be able to use the integral Universal Synchronous Asynchronous Receiver/Transmitter Port (USART) for asynchronous protocols.
- Understand the necessity for buffering long distance communication circuits.

Examine the parallel 3-digit 7-segment display interface of Fig. 11.13 on page 298 which uses both the parallel ports A and B. Although this is a working circuit, most of the parallel port budget of an 18-pin footprint device has been used up. Speed is certainly not a factor here, so a slower mode of data transmission is acceptable.

Consider the serial equivalent of Fig. 12.2. Here only two port pins are used. One labelled **SDO** (**Serial Data Output**) outputs the data bit by bit, most significant bit first. The other, labelled **SCK** (**Serial ClocK**) is used to clock the three shift registers at the one time, and hence shift the data right one bit at a time.

Each display has an associated 74HCT164 8-bit shift register[1] – see Fig. 2.20 on page 36. The 74HCT164 has a positive-edge triggered shift input clock C1 and two serial data inputs ANDed together at 1D. One of these data inputs can be used to gate the other input, but in our example they are both connected together to give a single serial input. There is also an active-low Reset input to clear the register contents, which are held high in the diagram. If desired, another port line can be used to drive R.

To change the display, a total of 24 bits will have to be shifted into the register array. To see how this can be done we will repeat the 7-segment

[1] All data outputs are simultaneously available and thus the 74HCT164 is described as a Serial-In Parallel-Out (SIPO) register as well as a SISO shift register.

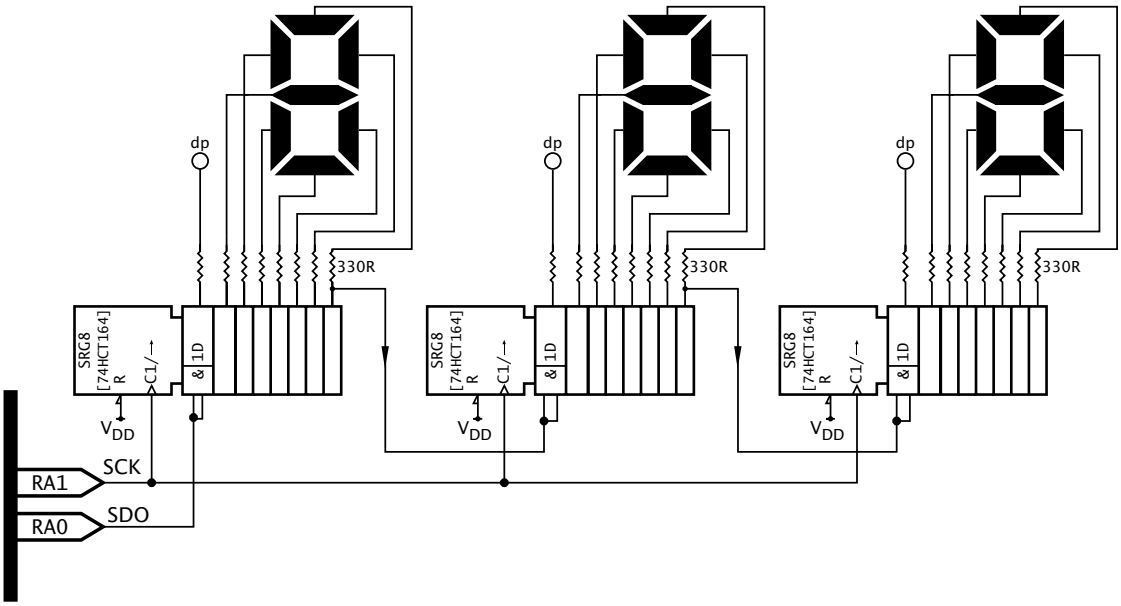


Fig. 12.2 Serial interface to a 3-digit 7-segment display.

driver routine of Program 11.7 on page 301 which converts a binary byte to an array of BCD digits in HUNDREDS, TENS and UNITS. These are mapped to 7-segment code and then sent out to each digit 8-bits at a time.

To serialize this process we require to design a subroutine to put each bit of a specified file register DATA_OUT out at SDO while pulsing SCK, beginning with the leftmost bit. A task list for such a subroutine is:

1. Bring SCK low.
2. COUNT = 8.
3. WHILE COUNT > 0 DO:
   (a) Shift DATA_OUT left into Carry.
   (b) Copy Carry to SDO.
   (c) Pulse SCK _/‾_.
   (d) Decrement COUNT.

Program 12.1 shows two subroutines. The first called DISPLAY is closely akin to Program 11.7 in that it calls the subroutines BIN_2_BCD and then sends the 7-segment coded bytes out to the interface registers. In this instance the units byte is sent first as this will eventually be shifted to the far end of the chain; followed by the tens and finally the hundreds byte.

The actual serial transmission is handled by the subroutine SPI_WRITE, which implements our task list. The datum placed by the caller in file register DATA_OUT is shifted left and the state of the Carry bit used to make the Serial Data Out pin RA0 0 or 1. The Serial ClocK pin RA1 is then toggled once _/‾_ to shift the data into the shift register chain. This is repeated eight times to complete the transaction, which takes a maxi-

Program 12.1 Displaying the decimal equivalent of a binary byte using a serial data stream.

```
SDO     equ    0
SCK     equ    1

DISPLAY bcf    PORTA,SCK    ; Initialize the clock line
        movf   BINARY,w     ; Get binary byte
        call   BIN_2_BCD    ; Convert to 3-digit BCD
        movf   UNITS,w      ; Get Units nybble
        call   SVN_SEG      ; Convert to 7-segment code
        movwf  DATA_OUT     ; Copy into the serial register
        call   SPI_WRITE    ; Shift it out

        movf   TENS,w       ; Get Tens nybble
        call   SVN_SEG      ; Convert to 7-segment code
        movwf  DATA_OUT     ; Copy into the serial register
        call   SPI_WRITE    ; Shift it out

        movf   HUNDREDS,w   ; Get Hundreds nybble
        call   SVN_SEG      ; Convert to 7-segment code
        movwf  DATA_OUT     ; Copy into the serial register
        call   SPI_WRITE    ; Shift it out
        return

; ****************************************************
; * FUNCTION: Clocks out a byte in series, MSB first  *
; * ENTRY   : Datum in DATA_OUT                       *
; * EXIT    : DATA_OUT zero                           *
; ****************************************************
; Task 1
SPI_WRITE
        bcf    PORTA,SCK    ; Make sure clock starts at low
; Task 2
        movlw  8            ; Initialize loop counter to 8
        movwf  COUNT
; Tasks 3(a)&(b)
LOOP    bcf    PORTA,SDO    ; Zero data bit
        rlf    DATA_OUT,f   ; Shift datum right into Carry
        btfsc  STATUS,C     ; Skip if Carry is 0
         bsf   PORTA,SDO    ; ELSE make data bit 1
; Task 3(c)
        bsf    PORTA,SCK    ; Pulse clock
        bcf    PORTA,SCK
; Task 3(d)
        decfsz COUNT,f      ; Decrement count
         goto  LOOP         ; and repeat until zero
        return
```

mum of 87 cycles to complete, depending slightly on the data pattern. A complete update of the display will take around 120 μs with a processor clock of 8 MHz and excluding the time spent in doing the data conversion.

Where a long chain of shift registers is being serviced, speed may be improved a little if each register has its own data feed but all clocked with the same SCK pin or sharing the same lines but each with a separate Enable. This latter technique is the method used in Fig. 12.8.

One problem with our shift register technique is that for the period where shifting is in process the data appearing at the port outputs are not valid; for 23 clock pulses in our example. Of course in this situation the response of the eye to microsecond changes in illumination makes this observation spurious. However, this may not always be the case and in such instances the shift register may be buffered from the parallel outputs using an array of D flip flops or latches, which can be loaded after the shifting process has been completed to give a single update.

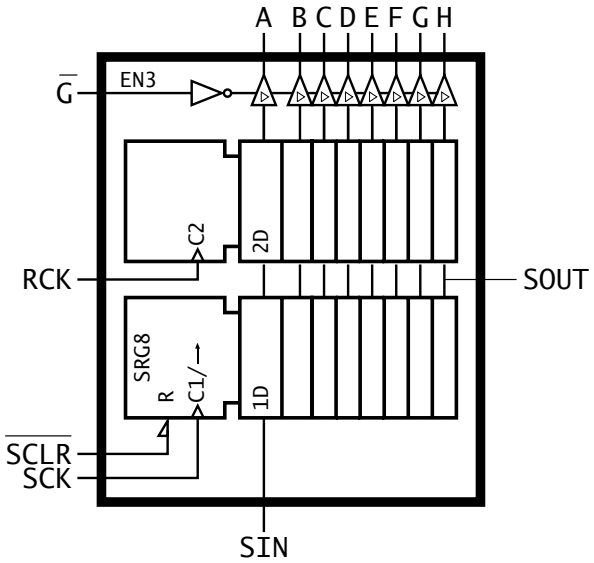


Fig. 12.3 Logic functional diagram of the 74HCT595 octal shift register with output register.

Rather than employing a separate buffer register, a more efficient solution typically uses the 74HCT595 of Fig. 12.3 with its integral 8-bit parallel-in parallel-out (PIPO) register between the shift register and the outside world. A rising edge _/‾ on the RCK (Register ClocK) pin transfers the serialized data to the parallel outputs. The last stage output of the shift register is made available to allow cascading to any length. All RCK pins can be pulsed together to allow the entire chain to simultaneously update.

One example where rippling of data may be undesirable, is were a digital datum is to be converted to its analog equivalent. In Fig. 12.4 the conversion is carried out using a National Semiconductor DAC0800. Essentially the analog voltage is a linear function of the 8-bit digital input

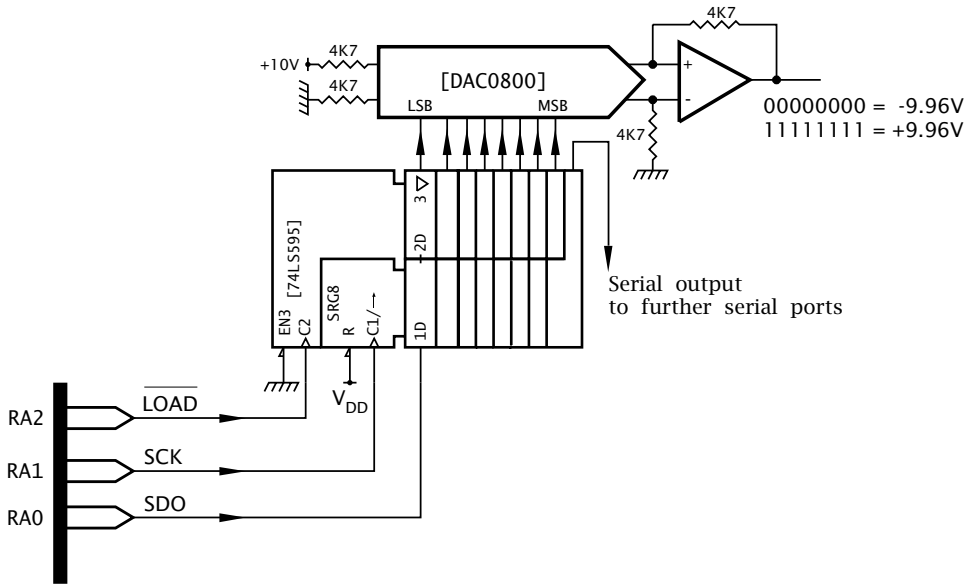


Fig. 12.4 Serially interfacing to a DAC0800 digital to analog converter.

varying from −9.96 V for an input of 00000000*b* through +9.96 V for 11111111*b* – see Fig. 14.13 on page 419.

Using a 74HCT595 registered shift register, the digital input does not change until the new datum is in place and the PIC pulses the C2 Register Clock, giving clean changes in the data presented to the DAC and corresponding analog output.

Data can be input serially in a similar manner using parallel-in serial-out (PISO) shift registers. The example shown in Fig. 12.5 is a serialized version of the intruder alarm of Fig. 11.10 on page 287 using only three lines to connect to all eight sensor groups; a considerable economy compared to the original 16 channels.

Each sensor group is attached to a 74HCT165 8-bit PISO shift register, with the serial output of the further register feeding the serial input of the next nearest register. Once the data has been loaded in, it may be shifted into the **SDI** (**Serial Data In**) parallel input RA1 and assembled bit by bit. In the specific case of the multi-zone intruder alarm, after each eight shifts the assembled byte can be tested for non zero and the appropriate action taken – see SAQ 12.1.

Also shown in Fig. 12.5 is the single output port used to display the active zone. As both input SDI and output SDO serial channels share the same shift clock SCK, then shifting data in will also clock this serial output port. Conversely, sending data to the output port will shift data in from the Zone ports. In this example there is no problem as microsecond fluctuations in the Zone lamps are of no consequence, and the sequence of operations ends with the output port being accessed with the earmarked data. Where this interaction is undesirable, then either the appropriate

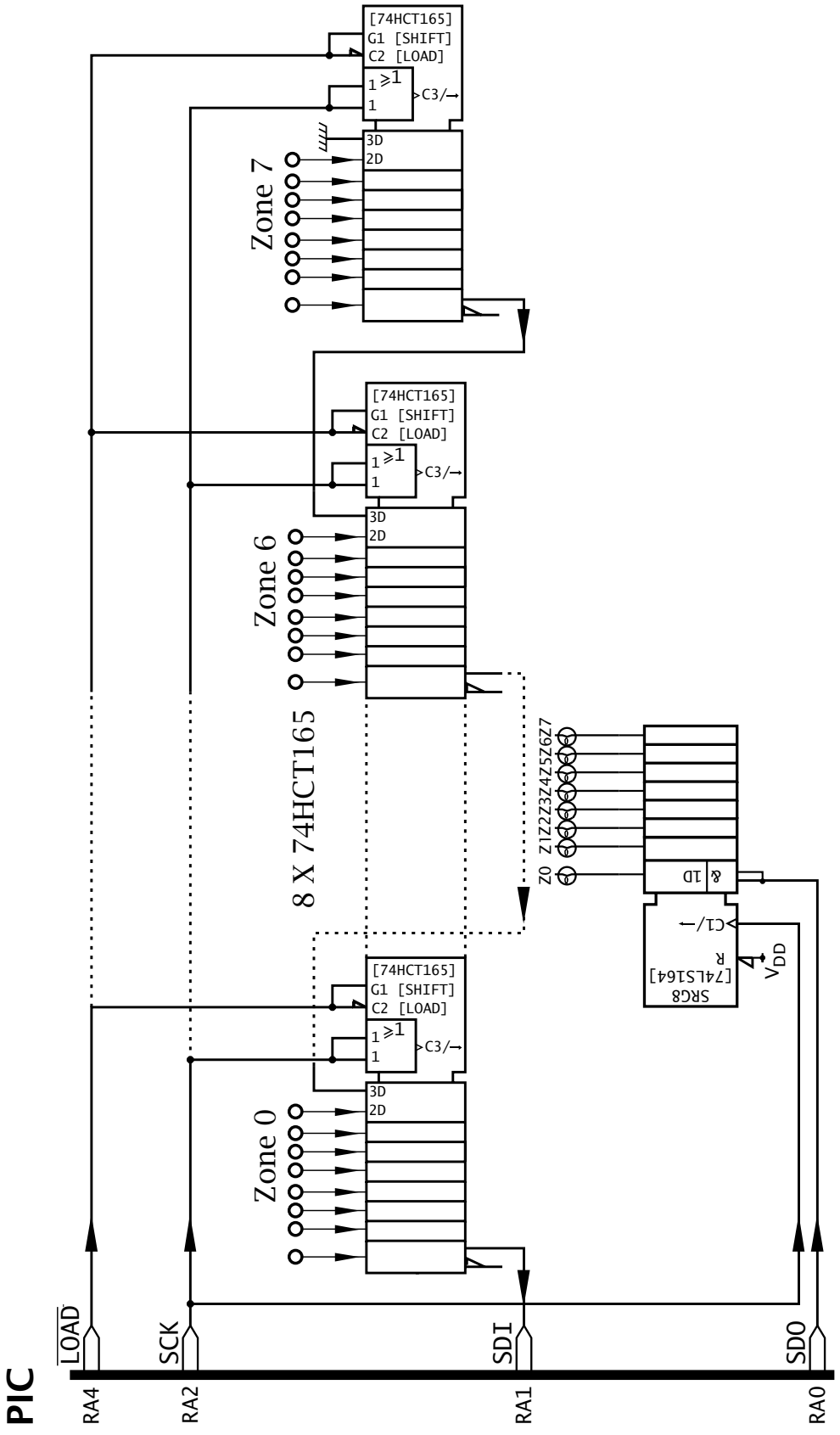


Fig. 12.5 Serially interfacing to the multi-zone intruder alarm.

datum should be continually presented to SDO at at the same time as it is read in at SDI or a latched register, such as the 74HCT595, used to staticize the display data. As an alternative, separate serial clock lines could be used.

The core serial interface software is the input counterpart of subroutine SPI_WRITE in Program 12.1, which implements the following task list:

1. Bring SCK low.
2. COUNT = 8.
3. WHILE COUNT>0 DO:
   (a) Pulse SCK _/‾_.
   (b) Copy input SDI to Carry.
   (c) Shift left Carry into DATA_IN.
   (d) Decrement COUNT.

This task list is similar to that for subroutine SPI_WRITE except that the Carry bit is shifted *into* the file register, its value depending on the state of the SDI pin following the clock pulse. After eight clock-shift-test loops the datum in DATA_IN is the parallelized byte assembled from the serial input port, with the first bit ending up in the leftmost significant placeholder in DATA_IN.

Program 12.2 Input serial byte subroutine.

```
; ******************************************************
; * FUNCTION: Clocks in a byte in series, MSB first    *
; * ENTRY   : None                                     *
; * EXIT    : Datum in DATA_IN; COUNT = 0              *
; ******************************************************
; Task 1: Bring SCK low
SPI_READ
        bcf    PORTA,SCK    ; Make sure clock starts at low
; Task 2: COUNT=8
        movlw  8            ; Initialize loop counter to 8
        movwf  COUNT
; Task 3: WHILE COUNT>0 DO:
; Task 3 (a): Pulse SCK
SER_IN_LOOP
        bsf    PORTA,SCK
        bcf    PORTA,SCK
; Task 3(b): Copy input SDI to Carry
        bcf    STATUS,C     ; Zero Carry
        btfsc  PORTA,SDI    ; Check data input
         bsf   STATUS,C     ; IF 1 THEN set Carry
; Task 3(c): Shift data bit in (Carry)
        rlf    DATA_IN,f    ; Shift data bit in (Carry)
; Task 3(d): Decrement COUNT and repeat Task3 WHILE>0
        decfsz COUNT,f      ; Decrement count
         goto  SER_IN_LOOP  ; and repeat until zero
        return
```

The subroutine coded in Program 12.2 is similar to the input subroutine of Program 12.1. Indeed they may be combined so that data is shifted out of a specified file register at at the same time as it is read in. This type of scheme is referred to as **full duplex**, as opposed to **half duplex** where only one direction at a time is possible. A serial link where data flow can only be in one fixed direction is known as **simplex**.

The serial protocol described in this example is commonly known as **Serial Peripheral Interface** (**SPI**).[2] Microwire is a similar, but not identical, serial protocol.[3] It is a sufficiently standardized protocol used by most microcontrollers to allow manufacturers to produce a range of peripheral devices specifically designed to directly interface to a SPI bus without the necessity to add external shift registers. As an example of this genre, the Maxim MAX549A is a dual digital to analog converter (DAC) which is powered with a V_{DD} of +2.5 V to 5.5 V. Its operating current is typically 150 μA per DAC at 5 V and either or both DACs can be shut down to reduce the current drain to less than 1 μs in its Standby mode. Data can be clocked in at a rate of up to 12.5 MHz. All this functionality is available in an 8-pin package, and should be contrasted with the 20-pin MAX506 of Fig. 14.12 on page 418 designed for direct parallel port connection.

A simplified functional diagram of the MAX549A is shown in Fig. 12.6. This shows an integral 16-stage shift register clocked from SCLK and fed data via DIN using the normal SPI protocol. The additional eight locations are used to store four control bits with the following functionality:

A0
Enables the input PIPO register for channel A and which is clocked on a rising edge at the $\overline{CE}$ pin.

A1
Enables the input PIPO register for channel B and which is clocked on a rising edge at the $\overline{CE}$ pin.

C1
Gates both DAC registers allowing them to be simultaneously updated by a _/‾ on $\overline{CE}$.

C2
When 1 will power down any DAC selected with A0 or/and A1. This disconnects the reference voltage V_{ref} from the DAC's resistor network (see Fig. 14.11 on page 416) and leaves only the residual current of less than 1 μA to activate the internal registers whose contents remain unchanged.

Both DACs have a 2-layer register pipeline isolating them from the shift registers. The first layer is the In registers, which are gated when A0 or A1 as appropriate is 1. The data sitting in the lower byte of the shift register can then be clocked in by pulsing $\overline{CE}$ (pin 3) low. This change will

[2] SPI is a trademark of Motorola Inc.
[3] Microwire is a trademark of National Semiconductor Corp.

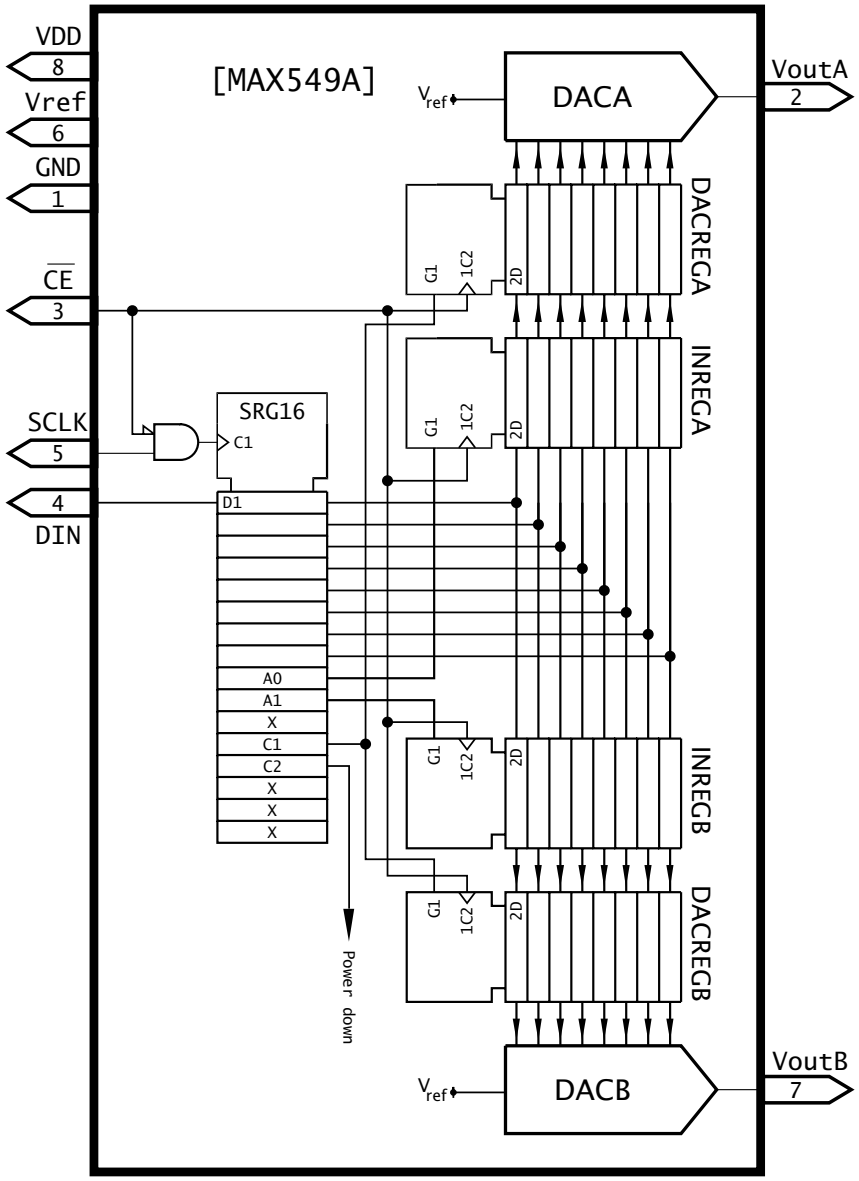


Fig. 12.6 The MAX549A SPI dual 8-bit DAC.

be stored but will not appear at the input of the DAC until the next layer PIPO register is clocked. This register is enabled when C1 is 1 and $\overline{\text{CE}}$ is pulsed. This means that one data byte can be sent to, say, DACA and then another to DACB. The DAC registers can then be updated together, resulting in both outputs V_{outA} and V_{outB} changing simultaneously; see Program 12.3. This can even be done when the MAX549A is asleep, as the registers are not affected by this power-down state.

From this discussion we see that each transition from the PIC takes two 8-bit transfers | Control | Data | followed by a $\_/\overline{\ }$ on the $\overline{\text{CE}}$ pin. For our example we will send the contents of File $20h$ to Channel A and then the contents of File $21h$ to Channel B, at that point updating both

DAC registers and hence outputting the analog equivalent of File $20h$ to pin V_{outA} and File $21h$ to pin V_{outB}. We assume that both AD0 and AD1 pins are connected to Ground.

Our implementation will involve the transmission of four bytes of information:

1. Control byte 1: XXX00X01
   No power down, update Channel A, no output change.
2. Data byte 1:
   Contents of File $20h$.
3. Control byte 2: XXX01X10
   No power down, update Channel B, both outputs change.
4. Data byte 2:
   Contents of File $21h$.

The hardware-software interaction is shown in Program 12.3. Four bytes are transmitted using subroutine SPI_WRITE, with the MAX549A's $\overline{CE}$ being pulsed _/‾_ after each [Control][Data] byte pair. The final

Program 12.3 Interacting with the MAX549A dual-channel SPI DAC.

```
       include  "p16f84.inc"
CE     equ 2

; ********************************************************
; * FUNCTION: Sends out Channel A & B data to SPI protocol *
; * FUNCTION: MAX549A simultaneously updating outputs      *
; * RESOURCE: Subroutine SPI_WRITE                         *
; * ENTRY   : Channel A in File20h, Channel B in File21h   *
; * EXIT    : Both analog outputs updated                  *
; ********************************************************
       movlw  b'00000001'  ; Control byte 1
       movwf  DATA_OUT     ; Put in designated location
       call   SPI_WRITE    ; and send out to MAX549A
       movf   20h,w        ; Get Channel A data
       movwf  DATA_OUT     ; Put in designated location
       call   SPI_WRITE    ; and send out to MAX549A
       bsf    PORTA,CE     ; Pulse CE
       bcf    PORTA,CE

       movlw  b'00001010'  ; Control byte 2
       movwf  DATA_OUT     ; Put in designated location
       call   SPI_WRITE    ; and send out to MAX549A
       movf   21h,w        ; Get Channel B data
       movwf  DATA_OUT     ; Put in designated location
       call   SPI_WRITE    ; and send out to MAX549A
       bsf    PORTA,CE     ; Pulse CE
       bcf    PORTA,CE
       return
```

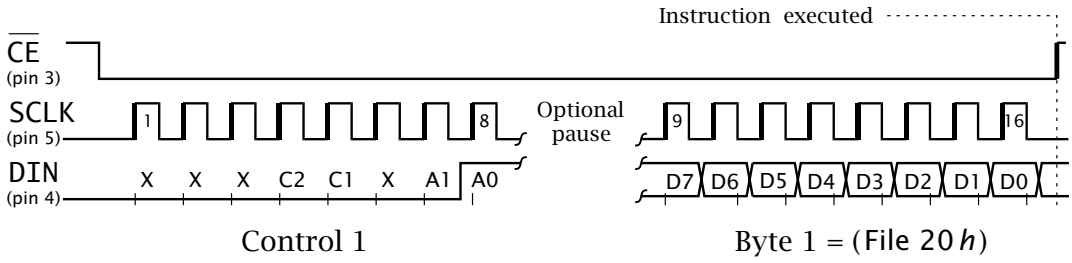


Fig. 12.7 SPI waveforms for the MAX549A.

process sets C1 high, which transfers both data bytes to the DAC registers at the same time updating the Channel B In register.

Looking at the three pins on the MAX549A would give a waveform similar to that of Fig. 12.7 for the transmission of the first | Control | Data | byte pair. During the transmission $\overline{CE}$ remains low with the data shifting into the MAX549A's integral shift register. After the second byte, i.e. the 16^{th} clock pulse, bringing $\overline{CE}$ high activates the selected internal registers, executing the instruction.

The diagram shows transitions on the DIN line from the PIC's SDO pin, occurring sometime before the active rising edge on SCK. Sometime is a vague term, obviously it must occur no later than a minimum time before _/‾ and be held for a short time after. The MAX549A data sheet gives the minimum set-up time t_{DS} of 30 ns and hold time t_{DH} of 10 ns. Even at a PIC clock rate of 20 MHz an instruction cycle takes 200 ns, so timing will not be violated.

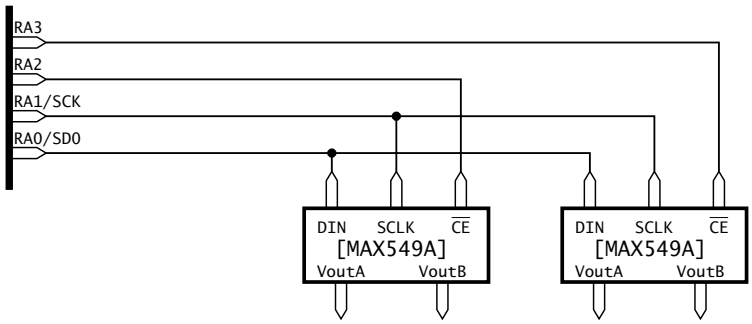


Fig. 12.8 Multiple MAX549As on the one SPI circuit.

By judicious use of the MAX549A's $\overline{CE}$ input, several DACs may be connected to the SCK/SDO lines with a serial transmission only being shifted into the device which has its $\overline{CE}$ low. Figure 12.8 shows two MAX549As sharing the one SPI link, giving four analog output channels in total. Us-

ing a 2 to 4-line decoder in conjunction with RA3:2 would enable up to four MAX549As with a total budget of only four port lines.

All 28-pin+ mid-range and high-range PICs feature integral ports which can be configured to conform to several serial protocols. A somewhat simplified representation of the basic serial module, such as provided on the PIC16C73/4, set up for the SPI protocol is shown in Fig. 12.9. The heart of the **Synchronous Serial Port** (**SSP**) is the Special Purpose Register file (SPR) **SSPBUF** (**SSP BUFfer**) at File 13*h*. A datum byte written into this SPR will automatically be transferred into the **SSP Shift Register** (**SSPSR**) and shifted out of the PIC's dedicated SDO pin, which is shared with Port C's RC5 I/O line. At the same time, eight bits of data will be shifted in from the SDI/RC4 pin. When this frantic burst of activity is completed, the new byte is automatically transferred to SSPBUF whence it can be read. This transfer is signalled by setting the **BF** (**Buffer Full**) flag in the **SSPSTAT** (**SSP STATus**) register at File 94*h* – shown in Fig. 12.10. In addition, the **SSP Interrupt flag SSPIF** (Peripheral Interrupt Register 1 – see Fig. 14.10 on page 408) is also set. Once SSPBUF is read, BF is automatically cleared. However, if using interrupts the SSPIF has to be 'manually' cleared in the ISR in the normal interrupt flag way.

The SSPSR can be clocked from four internal sources, which are selected by the programmer via the **SSPMode** bits **SSPM[1:0]** in the **SSP CONtrol** (**SSPCON**) register at File 14 h. Three of these frequencies are derived from the main PIC oscillator. For example, with a 20 MHz crys-

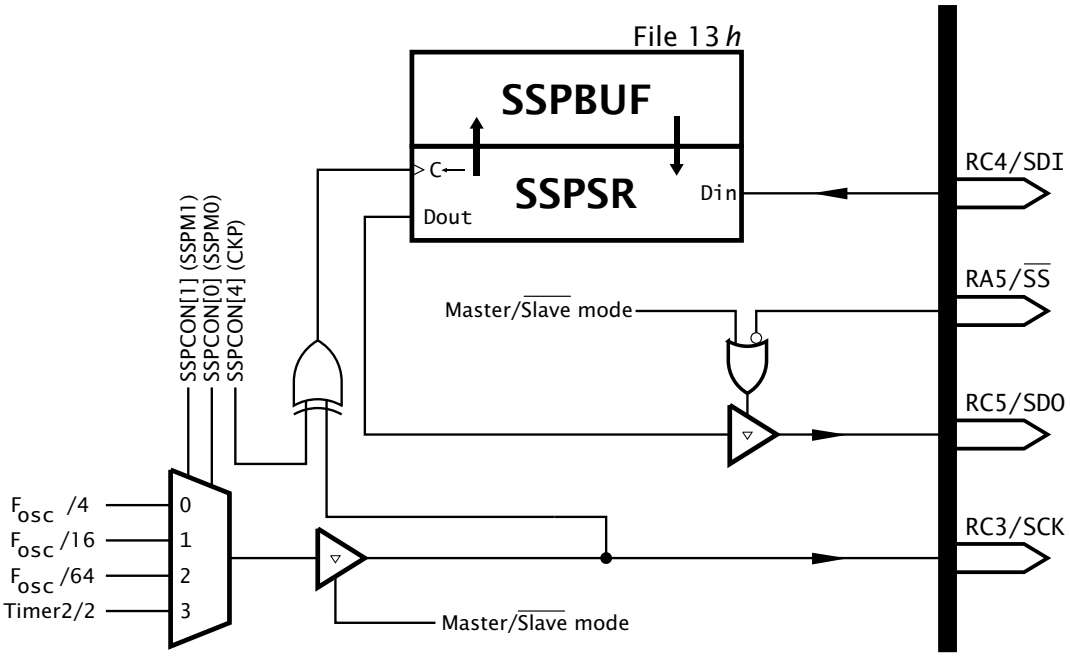


Fig. 12.9 The basic Serial Synchronous Port.

tal the SCK/RC3 shift rate can be selected as 5, 1.25 MHz and 312.5 kHz (200, 800 ns and 3.2 μs). The final selection gives the shift rate as half the frequency generated by Timer 2 overflowing – see Fig. 13.8 on page 379. This option is used where very slow shift rates are required.

As well as programmable frequency selection the idle polarity of the SCK may be set with the CKP bit at SSPCON[4]. With CKP = 0 the clock will idle low with valid data being made available on the ‾_ for an external shift register, as shown in Fig. 12.11.[4]

The two SSPMode combinations 0110*b* and 0101*b* (see Table 12.1) place the SSP in the **Slave** mode. As opposed to the **Master** mode, shifting is done using an external clock, usually generated from a remote Master device. In addition, when in Slave mode 0100*b* the PIC can become a listener only if its $\overline{SS}$ ($\overline{\text{Slave Select}}$) pin is high. This disconnects SDO and allows another Slave in a multidrop network to do the talking — see Fig. 12.12.

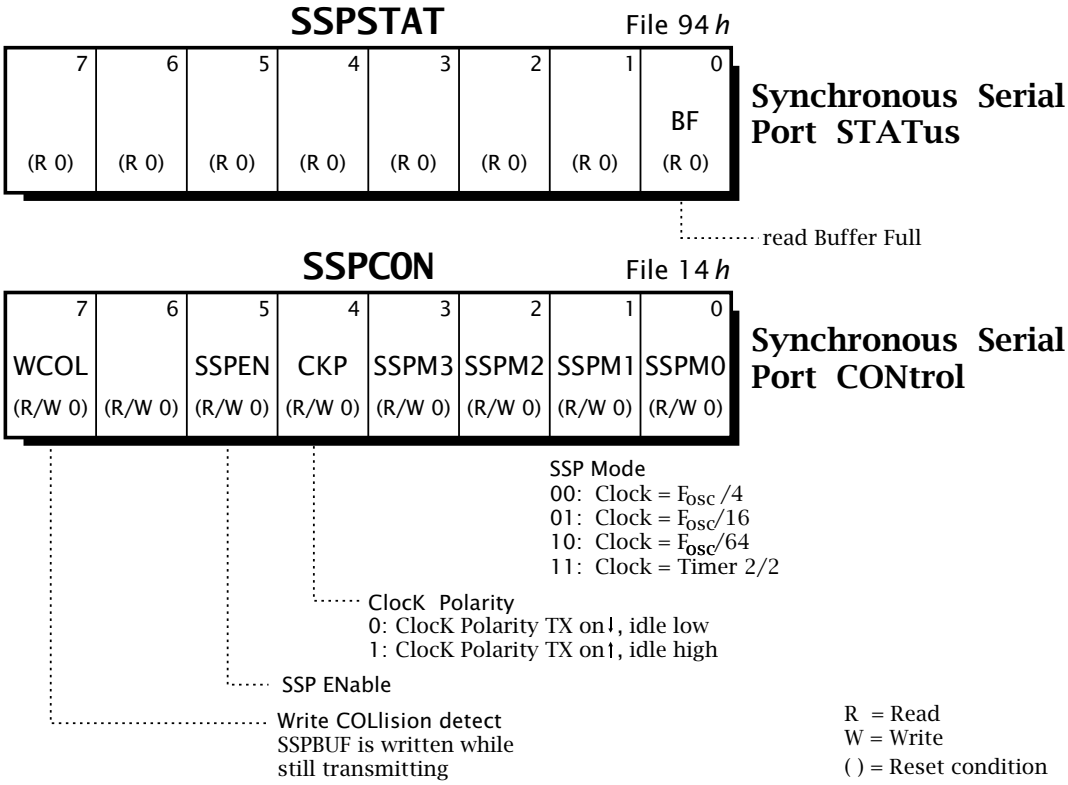


Fig. 12.10 The SSP CONtrol and STATus registers as appropriate to the SPI mode.

[4] A more advanced SSP port, such as used in the PIC16C774/F874, can control both active edge and idle polarity separately as well as input sampling time.

Figure 12.10 shows the SSP CONtrol register at File $14h$ and the associated SSPSTATus register in Bank 1 at File $94h$. The SSP port is enabled for whatever protocol when SSPEN (SSP ENable) in SSPCON[5] is 1. As SSPEN resets to zero, the SSP is disabled by default. In the disabled state the relevant Port C pins can be used as normal parallel I/O lines. If they are to be used as SSP lines then RC5:3 must be set via TRISC to be input or output as appropriate to their SSP function. Similarly, if the $\overline{SS}$ control is to be used, RA5 must be set to input.

Bits SSPCON[3:0] are the Mode control bits which set the communication protocol and various Master/Slave options as listed in Table 12.1. Of interest to us is the internal clock source in the SPI Master mode and the use of the $\overline{SS}$ pin when in the SPI Slave mode.

Finally, SSPCON[7] is the **Write COLlision** (**WCOL**) status bit – not in SSPSTAT due to lack of space. This is set to 1 if the software writes to SSPBUF *before* the transmission of the previous byte has been completed. If set, it should be cleared by software to reset this warning mechanism.

Using Figs. 12.9 and 12.10 as a programmer's model we can now deduce the hardware-software interaction in order to action a transmission of a byte and/or receive a new byte:

1. Configure SSP module.
   - Set up SCK/RC3, SDO/RC5 as outputs and SDI/RC4 as an input (TRISC[5:3])
   - Set up Master/Slave mode with appropriate clock source (SSPCON[3:0])
   - Choose active TX clock edge with CKP (SSPCON[4])
   - Enable the SSP by setting SSPEN (SSPCON[5])
2. Move datum to SSPBUF to initiate transmission.
3. IF WCOL = 1 THEN reset WCOL and go to item 2
4. Poll BF for 1 (SSPSTAT[0])
5. Move RX data from SSPBUF, which also resets BF

To illustrate this process, consider a subroutine SPI_IN_OUT which combines the function of SPI_READ and SPI_WRITE; that is it transmits the datum in file register DATA_OUT whilst at the same time returning the consequential received byte to DATA_IN.

Table 12.1: The SSP Mode bits.

SSPM[3:0]: Synchronous Port Mode select bits	
0000	SPI Master mode with SCK = $F_{osc}/4$
0001	SPI Master mode with SCK = $F_{osc}/16$
0010	SPI Master mode with SCK = $F_{osc}/64$
0011	SPI Master mode with SCK = TMR2 output/2
0100	SPI Slave mode. $\overline{SS}$ pin control enabled
0101	SPI Slave mode. $\overline{SS}$ pin control disabled
0110	I^2C Slave mode, 7-bit address
0111	I^2C Slave mode, 10-bit address
1011	I^2C Start & Stop bit interrupts enabled, Slave idle
1110	I^2C Slave mode, 7-bit address with Start & Stop interrupts enabled
1111	I^2C Slave mode, 10-bit address with Start & Stop interrupts enabled

The implementation of this subroutine depends on setting up the SSP during the initialization phase of the main software after Reset. In the following code fragment we are using the $F_{osc}/4$ clock rate Master mode:

```
        .include  "p16c74.inc"
MAIN    bsf   STATUS,RP0     ; Change to Bank 1
        movlw b'11010111'    ; RC5/SDO, RC3/SCK outputs
        movwf TRISC          ; RC4/SDI input
        ..... .....
        bcf   STATUS,RP0     ; Return to Bank 0
        movlw b'00100000'    ; Enable SSP, TX clock on -ve edge
        movwf SSPCON         ; SPI Master, Fosc/4 rate
```

The coding shown in Program 12.4 follows the task list exactly. Data to be transmitted is moved from the designated file register to SSPBUF and status bit WCOL checked to see that it got there. If there was a transmission in progress then the datum is not stored in SSPBUF and WCOL is set. If this subroutine is the only code to access the SSP then this should rarely be the case and in most instances this check is omitted, but its inclusion makes the system more robust.

Once the transmit datum is in situ, the transmit sequence is immediately initiated, as shown in Fig. 12.11 and progresses to its conclusion. Once the Buffer Full status flag BF is set, the received datum can be moved out of SSPBUF to its ordained location. This automatically resets BF.

Program 12.4 Using the SSP for SPI data input and output.

```
; ************************************************************
; * FUNCTION: Transmits and simultaneously receives one byte *
; * FUNCTION: from the SSP using the SPI protocol            *
; * ENTRY   : Data to be transmitted is in DATA_OUT          *
; * EXIT    : Data received is in DATA_IN                    *
; ************************************************************
SPI_IN_OUT
    movf   DATA_OUT,w       ; Get datum for transmission
    movwf  SSPBUF           ; Put into SSPBUF
SSP_IN_OUT_LOOP
    btfss  SSPCON,WCOL      ; Did it make it?
     goto  SPI_IN_OUT_CONT  ; IF so THEN continue
    bcf    SSPCON,WCOL      ; ELSE reset WCOL and try again
    goto   SSP_IN_OUT_LOOP
SPI_IN_OUT_CONT
    bsf    STATUS,RP0       ; Change to Bank0
    btfss  SSPSTAT,BF       ; Check for Buffer Full
     goto  SPI_IN_OUT_CONT  ; IF not then poll again
    bcf    STATUS,RP0       ; Back to Bank 1

    movf   SSPBUF,w         ; ELSE get the received datum
    movwf  DATA_IN          ; Put away
    return
```

Apart from a slight reduction in the code length, the advantage of using this hardware is the increase in speed. The actual transmit/receive takes eight SCK cycles, which in our case is eight instruction cycles. With an F_{osc} of 20 MHz the clocking rate is 5 MHz (that is a bit rate of 5 million bits per second, commonly written as 5 Mbit/s or 5 Mbps), giving a total time of 1.6 μs per byte.

If speed is of the essence then the SSP can be interrupt driven, as the SSPIF in the PIR1 register is set at the same time as BF. If the SSPIE mask bit in the PIE1 register is set, together with the overall mask bits GIE and PEIE in INTCON, then an interrupt will be generated when the outgoing byte has been transmitted and the new incoming byte is ready and waiting in SSPBUF. The function of these bits are shown in Fig. 14.10 on page 408. This interrupt can be used to awaken the PIC when in its Sleep mode.

Figure 12.11 shows the SPI mode timing for our subroutine. As we have cleared CKP then SCK is idling low. As soon as SSPBUF is written to, SCK goes high and the MSB of the TX datum appears at SDO. On the following edge, the MSB of the received datum is read in at SDI.

With this chosen polarity there is plenty of time for data from the external serial input port to present data to the PIC assuming that (as is usual) the its shift register is _/‾ triggered. This data is then sampled by the PIC on the following ‾_ of SCK, as indicated the the ↓ sample/shift points in Fig. 12.11. However, as we see, data that the PIC places on its SDO pin is placed on the _/‾ of SCK ready and stable for the following active ‾_. This means that the serial output port must be negative-edge triggered to ensure that it shifts in stable data. To get around this problem an inverter should be inserted at the peripheral's input shift register

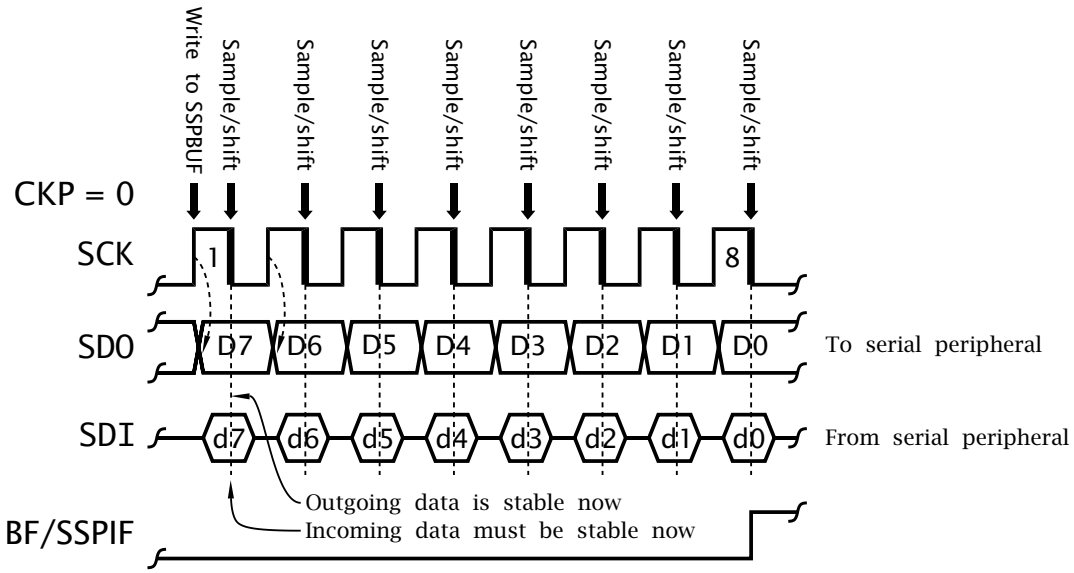


Fig. 12.11 SSP SPI-mode master waveforms.

clock, effectively converting it to a negative-edge triggered shift register. Some more advanced SSP modules, such as included in the PIC16F87X line, allow the input sample time to be shifted to remove the need for this additional hardware.

One use of serial transmission is to connect a number of PICs (or indeed other MCUs) together in one multiprocessor network. For example, a robot arm may have a MCU controlling each joint, communicating with a master processor. A simple multidrop circuit of one Master and two slave processors is shown in Fig. 12.12.

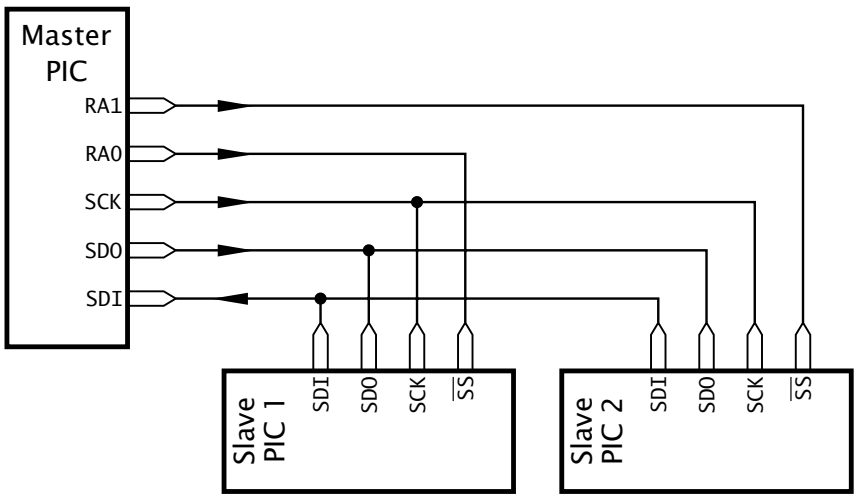


Fig. 12.12 A multidrop SPI communications network.

In this configuration the Master PIC externally drives the SCK of both Slaves, thus controlling when and how fast transmission occurs across the network. Both Slaves are configured in Mode 6 (SSPM[3 : 0] = 0110) so that the Slave Select inputs are enabled. Thus, if the Master wishes to read a datum from Slave 2 the the latter's $\overline{SS}$ is brought low and the Master clocks the eight bits from Slave 2's SSPBUF/SSPSR, into its own SSPBUF/SSPSR. At the same time any data transmitted by the Master will be received by the Slave. Slaves can rest in Sleep mode while all this is going on and the resulting interrupt used to awaken it after the transaction has completed provided that the PEIE and SSPIE mask bits are set.

SPI transactions may be coded in **C** either by mimicking the assemble-level code and setting/reading the appropriate registers, or by using built-in functions specific to the task. For example, for the CCS compiler constructions like:

```
spi_write(DATA_OUT);
DATA_IN = spi_read();
```

effectively mimic our SPI_IN_OUT subroutine.

To illustrate this technique, consider our interface to the MAX549A coded in Program 12.3. In order to do this the SSP needs to be configured using code of the form:

```
#include <16c74.h>
#bit     CE = 5.2      /* Port A, bit 2 to MAX549A's CE    */
void MAX549A(unsigned int channel_A, unsigned int channel_B);
set_tris_a(0xFB);      /* CE = RA2 output                  */
setup_spi(spi_master|spi_l_to_h|spi_clk_div_4);
```

has already been executed.

The MAX549A's $\overline{CS}$ is connected to Port A's RA2 pin, and is identified in the setup code using the #bit directive, as described on page 244. With this definition in mind, the code in Program 12.5 simply comprises four spi_write() calls with CE being pulsed between Control | Data pairs. The function may be called with an evocation something like MAX549A(data_x, data_y);

Program 12.5 Interfacing to the MAX549A in **C**.

```
void MAX549A(unsigned int channel_A, unsigned int channel_B)
{
spi_write(0x01);          /* Send out Control 1              */
spi_write(channel_A);     /* Send out Data 1                 */
CE=0;                     /* Pulse CE                        */
CE=1;
spi_write(0x0A);          /* Send out Control 2              */
spi_write(channel_B);     /* Send out Data 2                 */
CE=0;                     /* Pulse CE                        */
CE=1;
}
```

The key CCS compiler internal functions used for the SSPort in its SPI mode are:

setup_spi(spi_master|spi_h_to_l|spi_clk_div_4);
This function configures the SSP as an SPI Master, with clock polarity rising edge and a ÷4 clock frequency. These scripts, and others such as spi_slave are part of the included header file 16c74.h. This function also sets the direction of the appropriate Port A and Port C pins.

spi_write(value);
Used to write out the value from the SSP. Checks that the BF flag is set before returning.

spi_read();
Virtually identical to spi_write() except that it returns the value read by the SSP. If a value is passed to this function then it will be clocked out of SDO.

spi_data_is_in();
Returns non zero if a datum has been received over the SPI connection, that is if BF is set.

Although the SPI protocol is relatively fast, it requires a minimum of three data lines plus one select line for each Slave device. Apart from the cost, adding a device to an original design will require some hardware modification. By increasing the intelligence of the Slave device, it is possible to send both control, address and data in the one serial stream. The **Inter-Integrated Circuit** (**I²C**)[5] protocol developed by the Philips/Signetics Corporation in the early 1980s embodies this concept and also reduces the interface to only two lines by permitting bi-directional transmission.

SCL
The clock line sychronizing data transfer, serving the same function as SCK in the SPI protocol. However, SCL is bi-directional to allow more than one Master to take control of the bus at different times.

The original I²C specification set an upper limit on shift frequency of 100 kHz, that is 100 kbit/s, but the specification was augmented in 1993 with a Fast mode with an upper data rate of 400 kbit/s, which is the current de facto standard. In 1998 a compatible High-Speed mode was added with an upper bit rate of 3.4 Mbit/s.

SDA
The I²C data line allows data flow in either direction. This bi-directionality allows communication from Master to Slave (Master-Write) or from Slave to Master (Master-Read). Furthermore it allows the receiver to signal its status back to the transmitter at the end of each byte.

The I²C protocol is relatively complex and its full specification can be viewed at the Philips/Signetics Corporation web site[6]. Before looking at the basic protocol, we need to examine the SCL and SDA lines in more detail. When no data is being transmitted, both lines should be high; the **Idle condition**. A device wishing to seize control of an idling bus must bring its SDA output low. This is known as the **Start condition**. In order for the would-be Master to be able to pull this line low all other devices hung on the line must have their SDA pins open circuit and the line as a whole pulled up high through a single external resistor – see Fig. 12.14(a). To implement this SDA (and also SCL) outputs must be open-collector or open-drain – see Fig. 2.2(b) on page 19. This means that any device hung on the bus is able to pull its line low by outputting a logic 0.

After the Start condition the Master then sends eight bits of information synchronized with eight clock pulses. During this period any changes in data on SDA *must* occur when the clock is low. Data can then

[5] I²C is a trademark of the Philips Corporation.
[6] www.semiconductors.com/acrobat/various/I2C_BUS_SPECIFICATION_3.pdf

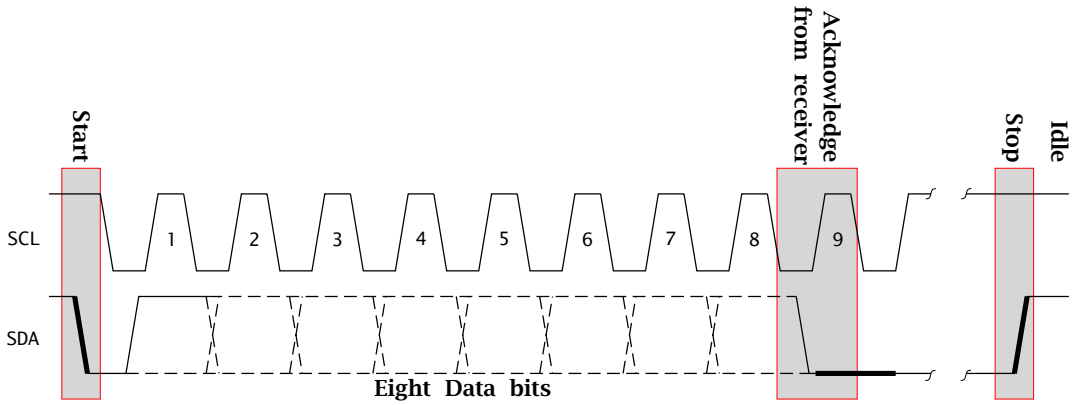


Fig. 12.13 Data transfer on the **I²C** bus.

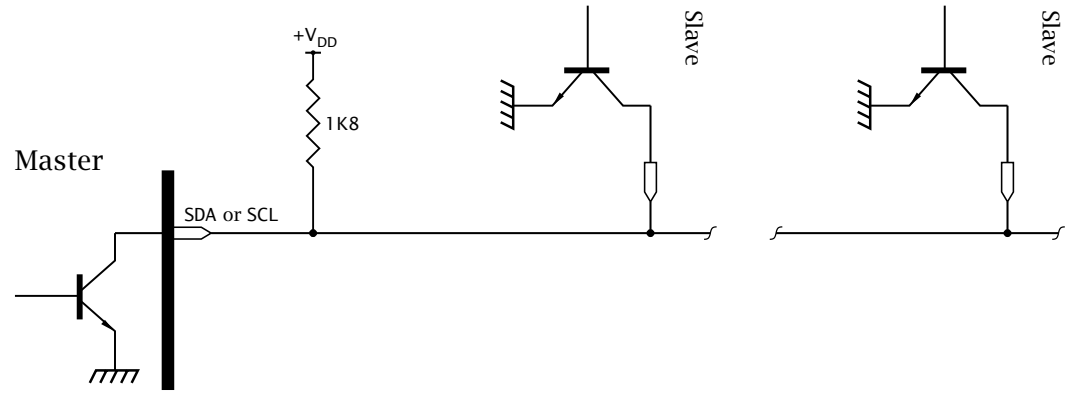
be clocked into the slave on the following SCL rising edge. This data may represent address, control or information bits.

At the end of the byte transfer, the Slave pulls SDA low to tell the Master that the data byte has been received. To allow this to happen the Master must release its control over SDA by allowing it to be pulled high via the bus resistor. Should the Slave not **Acknowledge** the transfer then the Master should abort the transfer and usually try again until timed out.

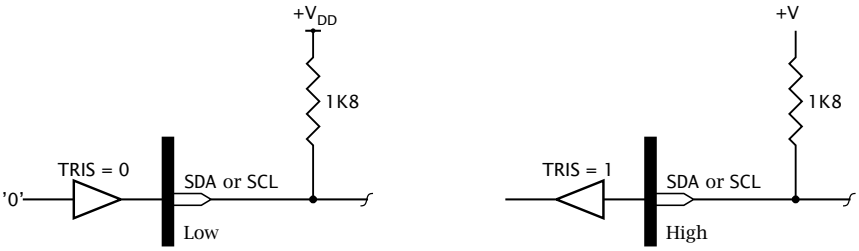
Several byte transfers will occur before the Master completes the transmission by bringing SDA high when the Clock is high. This **Stop condition** terminates the conversation. Another one can be initiated during the subsequent Idle state when the Master again sends out a Start bit. A Slave can distinguish the special Start and Stop situations, as this is the only time SDA changes when SCL is high.

The data transmission is framed between Start and Stop pairs and data flow can either be Master-Write as described or from Slave to Master in Master-Read mode. In the latter case it is the Master that acknowledges each byte. If it does not (No ACKnowledge or NACK) then the Slave knows that the Master has finished and floats its SDA pin allowing the Master to output a Stop bit and then Idle both lines – see Fig. 12.23(b). It is possible for the Master to begin by writing data to the Slave and then request a change of direction. In this case a second Start bit is sent by the Master without stopping, followed by the appropriate control bits.

In using a PIC MCU to implement the I²C standard a problem arises as port outputs are not open-drain; that is the logic 1 output state is not open-circuit as required in Fig. 12.14(a). However, it is possible to get around this, simulating the high impedance state by switching the port line output to input. For example, if we wish to use RA2 as the SCL data line then to pulse SCL low and then high we have:



(a) Connection of I^2C devices to the I^2C bus.



i Output is low *ii Output is pulled high*

(b) Using the PIC to simulate open collector.

Fig. 12.14 Sharing the SCL and SDA bus lines.

```
bcf  PORTA,2    ; Sometime during setup make PORTA[2] 0
.... .....
bsf  STATUS,RP0 ; Change to Bank1
bcf  TRISA,2    ; RA2 is output = 0
nop             ; Short delay
bsf  TRISA,2    ; Float RA1 by making it an input
bcf  STATUS,RP0 ; Return to Bank0
```

where the high state is a consequence of the external pull-up resistor and the high *input* impedance, as shown in Fig. 12.14(b)*ii*.

A complete transmission between Master and Slave comprises a packet of several byte/Acknowledge transfers sandwiched between a Start and Stop condition. To some extent the form of this packet depends on the requirements of the Slave device; however, all packets conform to the general sequence Slave address:Control/Command:Data shown in Fig. 12.15.

The essence of the I^2C protocol is the requirement that each type of Slave device has an address. This address is allocated[7] to the man-

[7] By the I^2C-bus committee.

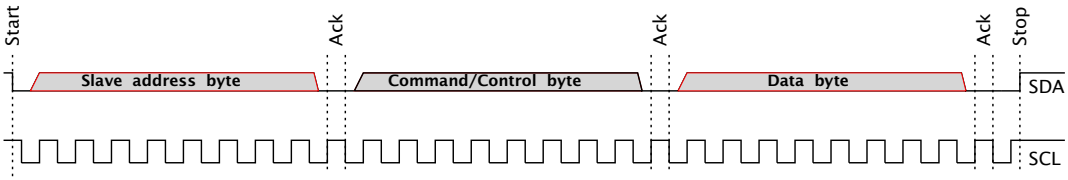


Fig. 12.15 A **I²C** packet transmission.

ufacturer of the I²C peripheral and is factory programmed. To allow more than one device of the same kind to share the same bus, most I²C-compatible devices allow two to four bits of this address to be set locally by the designer, usually by connecting Slave address pins to the appropriate logic levels. On receipt of a Start bit, all Slaves on the bus will examine the first seven bits for their personal address. If there is no match then the rest of the conversation is ignored until the next Start bit. Bit 8 is a direction bit, R/$\overline{\text{W}}$ is low if the Master is to be the transmitter and high if the Slave is to be the transmitter.

Not all 7-bit addresses are valid. All addresses matching 0000XXX*b* or 1111XXX*b* are reserved for special situations, leaving 224 valid addresses in total. Along with the introduction of a Fast mode, the I²C protocol was extended to permit a 10-bit address. This is signalled by the reserved address 11110XX*b*. In this case a second address byte is sent with the two XX bits in the first byte making up the total 10-bit Slave address.

After the address byte(s), the next byte is usually treated by the addressed Slave as a Command/Control word, passing configuration information. For example a I²C memory may require the internal address where the data is to be written to – see Example 12.3. Bytes following this are usually pure data or a mixture of data and control bytes.

In order to illustrate these concepts we will use the Maxim MAX518 DAC, shown in Fig. 12.16, as our exemplar. This is the I²C counterpart to the SPI protocol MAX549, with a 2-layer register pipeline, two channels and a power-down feature.

The MAX518 has a 7-bit Slave address of the form **01011**AD1AD0 where AD1 and AD0 are the logic state of pins 5 and 6 respectively. If we assume that both pins are connected to GND then the Address byte sent out by the Master will be | 01011 | 00 | 0 | with R/$\overline{\text{W}}$ being 0, as this device can only be written to.

The Command byte is of the form 000 RST PD XX A0 with three active control bits:

A0
Enables the input PIPO register for Channel 0 if 0 and Channel 1 if 1.

PD
When 1 will power down both DAC channels, reducing the supply current to typically 4 μA. The contents of the internal registers remain unchanged

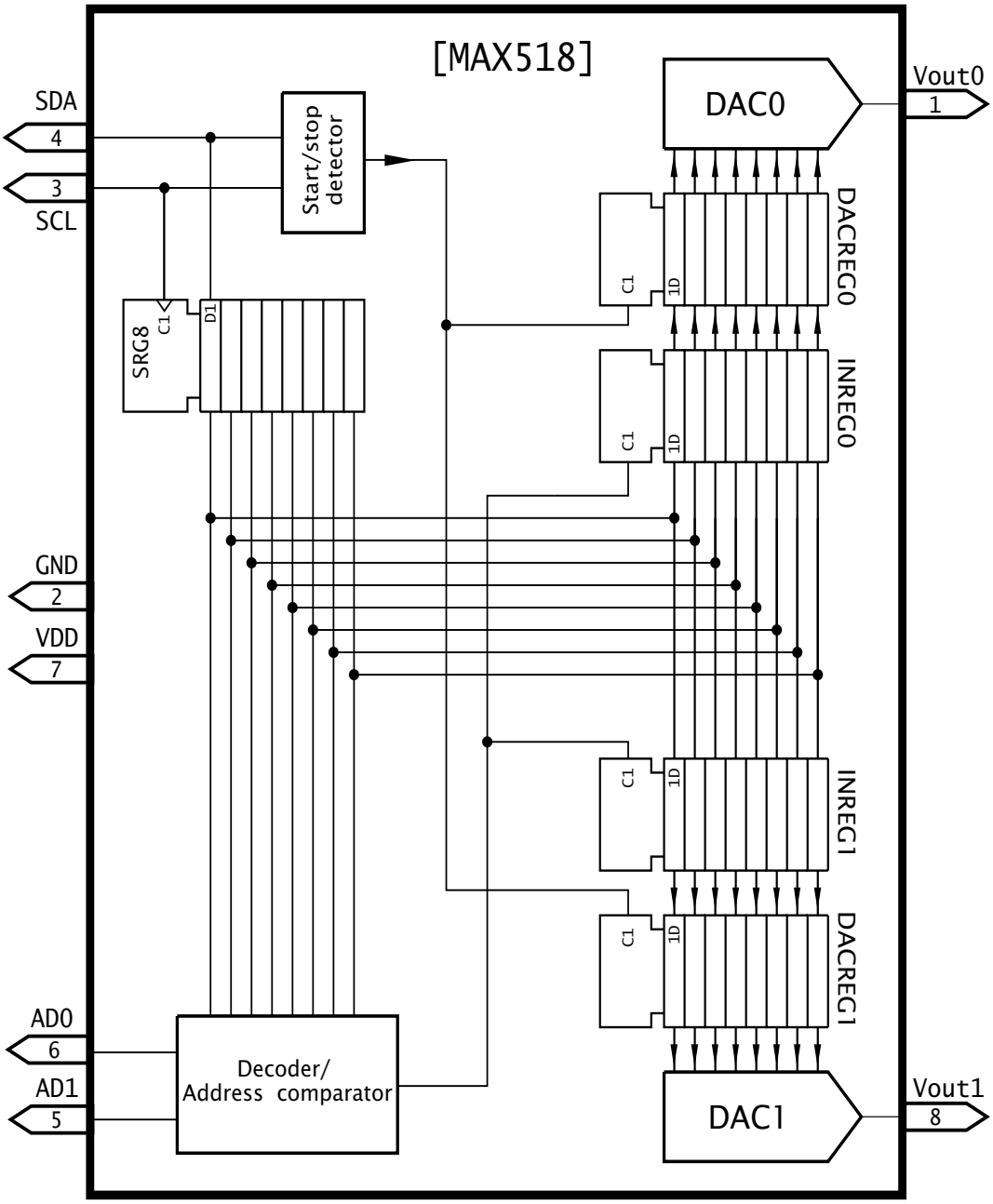


Fig. 12.16 The MAXIM MAX518 **I²C** dual digital to analog converter.

and data may be shifted in and registers updated in this condition. The state information is only executed whenever a Stop condition is sent by the Master, when the last transmitted value of PD is acted upon.

RST

All internal registers are cleared irrespective of the following data byte which may be treated as a dummy byte. Analog outputs go to zero after the Stop condition.

In all cases the Stop condition updates the analog outputs according to the commands and data byte. If there have been several Command:Data byte pairs since the last Stop then the most recent command and data are reflected in the state and output of the device.

In order to interface to the MAX518 we will need to design subroutines to send out a Start condition, a Stop condition and a Master-Write byte. See Program 12.14 for a Master-Read subroutine. To design the device driver we need to look more closely at the time relationship between Clock and Data signals, which generally are more tightly defined than in the SPI protocol.

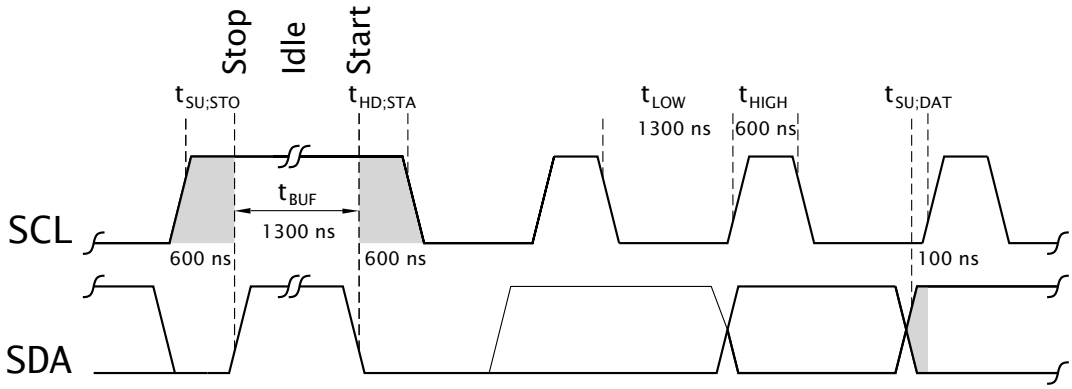


Fig. 12.17 Minimum timing relationships for the Fast **I²C** mode.

The MAX518 and most current I²C-compatible devices are designed to the Fast mode specification and the figures used in Fig. 12.17 relate to this 400 kHz clocking rate. Of particular note is the requirement that the clock SCL should be held high not less than 0.6 μs ($t_{HD;STA}$) after the active ¯_ of SDA to signal a Start condition. Similarly, a Stop condition requires that the clock be set up high at least 0.6 μs ($t_{SU;STO}$) before the active _/¯ of SDA. A minimum of 1.3 μs is required with the bus free (t_{BUF}) in the Idle state between a Stop and a following Start condition. These requirements allow time for the Slave devices to detect these synchronizing events without ambiguity.

During a data byte transmission the clock should be low no less than 1.3 μs (t_{LOW}) and high no less than 0.6 μs within the 2.5 μs overall duration limitation imposed by the 400 kHz. Data changes only when the clock is low, and any change should be complete no less than 100 ns ($t_{SU;DAT}$) before the clock goes high.

Not shown in the diagram is the maximum rise and fall times which should not exceed 300 ns with a maximum bus capacitance of 400 pF. To keep within this transition restriction the pull-up resistors of Fig. 12.14 should not be more than 1.8 kΩ with this value of capacitance. With short

bus runs and few Slave devices this value of resistance can be increased by up to a factor of ten to reduce energy dissipation when an output pin is low.

In implementing the I^2C timings, a PIC with a crystal above 3.2 MHz, with an execution time of less than 1.25 μs, may need to insert short delays between actions. For example, a 20 MHz crystal driven PIC implementing the instruction pair:

```
    bcf  TRISA,SCL
; Drag Clock low by making pin an output to logic 0
    bsf  TRISA,SCL
; Float clock high by making pin an input
```

would give high and low durations of only 0.2 μs. Short delays are conveniently implemented using `nop` (No OPeration) instructions; each taking one instruction cycle ($F_{osc}/4$). For example, to give a nominally 400 kHz clock at 20 MHz we have:

```
    bcf  PORTA,SCL   ; Clock low
    nop              ; 0.2us
    nop              ; 0.4us
    nop              ; 0.6us
    nop              ; 0.82us
    nop              ; 1.0us
    nop              ; 1.2us
    bsf  PORTA,SCL   ; Clock high
    nop              ; 1.6us
    nop              ; 1.8us
    nop              ; 2.0us
    nop              ; 2.2us
    nop              ; 2.4us
    nop              ; 2.6us
```

Of course slower clock speeds require less `nop`s but rather than tailor our subroutines for one particular crystal we will use the assembler macro called `Delay_600`, coded in Program 12.6, that will expand to the appropriate number of `nop`s to give a nominal 600 ns (0.6 μs) delay, depending on the value of the constant `XTAL` defined by the programmer at the head of the source file.

For example to alter the coding of Program 12.7 to suit a 12 MHz crystal system then the one line `#define XTAL 20` should be altered to `#define XTAL 12` and the program reassembled.

The coding of Program 12.6 makes use of the conditional assembly directive `if – endif`. This is similar to the **C** language statement `if(true){do this;}` of page 249 in that all instructions down to the following `endif` are implemented if the argument of the `if` directive is true. For example, `if((XTAL>6)&&(XTAL<=13))` states that if the constant `XTAL` is greater than 6 AND less than or equal to 13 then insert

Program 12.6 A crystal frequency-independent short delay macro.

```
Delay_600 macro             ; Delays by nominally 0.6us
          if (XTAL <= 6)
          nop               ; One nop if XTAL is less than 6MHz
          endif
          if ((XTAL > 6) && (XTAL <= 13))
           nop              ; Two nops delays if
           nop              ; XTAL is between 6 & 13MHz
          endif
          if (XTAL > 13)
           nop              ; Three nop delays if
           nop              ; XTAL is above 13MHz
           nop
           endif
         endm
```

two nop instructions. At 13 MHz this will be approximately 600 ns. In practice, extra delays will be introduced by instructions toggling the bus lines and executing housekeeping tasks. Thus some fine tuning can be undertaken if maximum speed is a criterion.

Based on the macro of Program 12.6 and the following initialization code:

```
          include "p16f84a.inc"
          #define XTAL 20

SCL       equ     0
SDA       equ     1

MAIN      movlw   TRISA       ; Set up the File Select Register
          movwf   FSR         ; to point to TRISA
          bcf     PORTA,SCL   ; Preset Clock & Data pins to 0
          bcf     PORTA,SDA   ; so that line can be dragged low
          bsf     INDF,SCL    ; Float Clock line high
          bsf     INDF,SDA    ; and the Data line to Idle state
```

which assumes that we are using Port A bits 0 and 1 of a 20 MHz PIC16F84A to implement our SCL and SDA lines, we can code the three subroutines outlined in Program 12.7 to allow us to communicate with the I^2C MAX518.

START

This subroutine releases both the SCL and SDA lines which are then pulled high to ensure the bus is in its Idle state for the minimum duration 1.3 μs t_{BUF}. Bringing SDA low gives the characteristic Start ¯_, which is followed by a 0.6 μs delay to implement $t_{HD;STA}$ (see Fig. 12.17) before the subroutine exits with both SCL and SDA low.

Program 12.7 Low-level **I²C** subroutines.

```
; ************************************************************
; * FUNCTION: Outputs the Start condition                    *
; * ENTRY   : FSR points to the I2C port's TRIS register     *
; * EXIT    : Start condition and SCL, SDA pins low          *
; ************************************************************
START     bsf   INDF,SDA      ; Ensure that we start with the
          bsf   INDF,SCL      ; Data and Clock lines pulled hi
          Delay_600           ; 1.3us delay in Idle state
          Delay_600
          bcf   INDF,SDA      ; Low-going edge on Data line
          Delay_600           ; Wait for Slave to detect this
          bcf   INDF,SCL      ; Exit with the Clock line low
          return

; ************************************************************
; * FUNCTION: Outputs the Stop condition                     *
; * ENTRY   : FSR points to the I2C port's TRIS register     *
; * EXIT    : Stop condition and SCL, SDA pins high (Idle)   *
; ************************************************************
STOP      bcf    INDF,SCL     ; Make sure that Clock line is low
          bcf    INDF,SDA     ; and the Data line is low
          bsf    INDF,SCL     ; Bring Clock line high
          Delay_600           ; for a minimum of 0.6us
          bsf    INDF,SDA     ; Rising edge on Data signals Stop
          return              ; including the return time

; ************************************************************
; * FUNCTION: Transmits byte to Slave and monitors Acknowledge*
; * ENTRY   : 8-bit data to be TXed is in DATA_OUT           *
; * RESOURCE: START and STOP subroutines                     *
; * EXIT    : Byte transmitted. ERROR is 01 IF no Ack received*
; * EXIT    : from Slave ELSE 00. SCL low                    *
; ************************************************************
I2C_OUT   bcf    INDF,SCL     ; Make sure that Clock line is low
          clrf   ERR          ; Start with no error
          movlw  8            ; Loop counter = 8
          movwf  COUNT

I2C_OUT_LOOP
          bcf    INDF,SDA     ; Data bit low?
          rlf    DATA_OUT,f   ; Shift data left once into Carry
          btfsc  STATUS,C     ; Is C 0 or 1
           bsf   INDF,SDA     ; IF the latter THEN make Data hi
          Delay_600           ; Delay plus xtra instructions OK
          Delay_600
          bsf    INDF,SCL     ; Bring Clock pin high
          Delay_600           ; for at least 0.6us
          bcf    INDF,SCL     ; Bring Clock low
          decfsz COUNT,f      ; Decrement loop count
           goto  I2C_OUT_LOOP ; and repeat eight times

; Now check Acknowledge from Slave
          bsf    INDF,SDA     ; Release Data line
          Delay_600           ; Keep Clock line low
          Delay_600           ; long enough for Slave to respond
          bsf    INDF,SCL     ; Bring Clock line high
          btfsc  INDF,SDA     ; Check if Data is low from Slave
           incf  ERR,f        ; IF not THEN ERROR1
          bcf    INDF,SCL     ; Now finish ACK by bringing Ck lo
          return
```

STOP

The Stop condition is implemented by ensuring that both SCL and SDA lines are low (which should be the case after an Acknowledge condition) and then releasing the SCL line which is then pulled high. After a $0.6\,\mu$s delay to implement $t_{SU;STO}$ SDA is released to give the characteristic Stop _/‾. The subroutine exits with both lines released and the bus Idling in preparation for the next Start condition.

I2C_OUT

This subroutine clocks out the eight bits placed in DATA_OUT by the caller, MSB first, and then checks that the Slave has Acknowledged the transaction.

The first part of this process is implemented by repetitively shifting the datum in DATA_OUT and inspecting the Carry flag. SDA is set to mirror **C** and the SCL line toggled to accord with the t_{LOW} and t_{HIGH} parameters illustrated in Fig. 12.17.

Once the loop count reaches zero, the Data line is released with SCL low for the duration t_{LOW}. SCL is then released high and the state of SDA, which should have been dragged low by the Slave, checked. If not low, the No ACKnowledge (NACK) situation is returned with ERR = $01h$; otherwise it will be zero.

Our use of errors here is very rudimentary. For instance, errors can also occur if some other device has locked either line low; that is the bus is busy.

We have not coded a Master-Receive I^2C counterpart to subroutine I2C_OUT, as the MAX518 only demands a Master-Transmit data interchange. However, Program 12.14 gives the I2C_IN mirror.

As our example we will send the contents of File $20h$ to the MAX518 Channel 0 and then the contents of File $21h$ to Channel 1; at that point updating both DAC registers and hence simultaneously outputting the analog equivalent of File $20h$ to pin V_{out0} and File $21h$ to pin V_{out1}. We assume that both AD0 and AD1 pins are connected to Ground.

Our implementation will involve the transmission of a packet of five bytes of information sandwiched between a Stop and a Start condition.

1. Start condition.
2. Address byte: $01011000b$
   Slave address 01011(00), Write.
3. Command byte 1: $00000XX0b$
   No ReSeT, no Power Down, Channel 0.
4. Data byte 1:
   Contents of File $20h$.
5. Command byte 2: $00000XX1b$
   No ReSeT, no Power Down, Channel 1.
6. Data byte 2:
   Contents of File $21h$.
7. Stop condition.

Program 12.8 Interacting with the MAX518 dual-channel **I²C** DAC.

```
ANALOG    call    START       ; Start a transmission packet
; Address byte
          movlw   b'01011000'; Slave address Master-Write
          movwf   DATA_OUT    ; Copied to pass location
          call    I2C_OUT     ; Send it out
          movf    ERR,f       ; Check for an error
          btfsc   STATUS,Z    ; IF Zero THEN continue
           goto   ANALOG      ; ELSE try again
; Command byte 1
          movlw   b'00000000'; No ReSeT, No Power Down, Channel0
          movwf   DATA_OUT    ; Copied to pass location
          call    I2C_OUT     ; Send it out
          movf    ERR,f       ; Check for an error
          btfsc   STATUS,Z    ; IF Zero THEN continue
           goto   ANALOG      ; ELSE try again
; Data byte 1
          movf    20h,w       ; Channel0's datum from memory
          movwf   DATA_OUT    ; Copied to pass location
          call    I2C_OUT     ; Send it out
          movf    ERR,f       ; Check for an error
          btfsc   STATUS,Z    ; IF Zero THEN continue
           goto   ANALOG      ; ELSE try again
; Command byte 2
          movlw   b'00000001'; No ReSeT, No Power Down, Channel1
          movwf   DATA_OUT    ; Copied to pass location
          call    I2C_OUT     ; Send it out
          movf    ERR,f       ; Check for an error
          btfsc   STATUS,Z    ; IF Zero THEN continue
           goto   ANALOG      ; ELSE try again
; Data byte 2
          movf    21h,w       ; Channel1's datum from memory
          movwf   DATA_OUT    ; Copied to pass location
          call    I2C_OUT     ; Send it out
          movf    ERR,f       ; Check for an error
          btfsc   STATUS,Z    ; IF Zero THEN continue
           goto   ANALOG      ; ELSE try again

          call    STOP
```

The listing of Program 12.8 follows our itemization exactly. On return from each call to `I2C_OUT` the Error datum is tested for zero. If non zero then the process is restarted. Repeated Starts are allowed by the I²C protocol. However, if there was a hardware fault with the bus or Slave then this process would continue indefinitely. Thus, for robustness a time-out mechanism should be implemented to prevent hang-ups.

Although the basic Synchronous Serial Port fully implements most I²C Slave functions, there is little support for the role of Master.[8] However, if a PIC with an integral SSP is to be used as a Master then it is advantageous to use pins `RC3` and `RC4` for `SCL` and `SDA` respectively. With the SSP

[8] Some newer PICs, such as the PIC1687X line, have an enhanced Serial port with most Master functions available in hardware.

enabled and programmed in Mode 11 (Slave idle – see Table 12.1) these pins conform to the specified relatively slow (300 ns maximum) rise and fall times. A normal port line has transition times of the order of 10 ns. The slower transition times give less cross talk between bus lines and less transmission line reflections at electrical discontinuities.

As in the case for the SPI protocol, many **C** compilers targeted to the PIC have built-in functions to implement the I^2C protocol and avoid bit banging user-defined functions.

To illustrate the technique, consider Program 12.9 which replicates the assembly-level coding of Programs 12.7 and 12.8 using the CCS compiler.

`i2c_start();`
Generates the Master Start condition.

`i2c_stop();`
Generates the Master Stop condition.

`i2c_read();`
Reads a byte over the bus. If an optional parameter of 0 is used then the Master will not Acknowledge the received data.

Program 12.9 Interfacing to the MAX518 in **C**.

```
#include <16F84.h>
/* PortA, bit0 is the Master SCL, bit1 is the Master SDA,
   fast protocol                                          */
#use i2c(master, scl=PIN_A0, sda=PIN_A1, fast)
#define data_x *(unsigned int *)0x20
#define data_y *(unsigned int *)0x21

void MAX518(unsigned int channel_0, unsigned int channel_1);

main()
{
/*      Various code lines                                */
MAX518(data_x, data_y); /* Send out the two data bytes   */
/*      More code                                         */
}

void MAX518(unsigned int channel_0, unsigned int channel_1)
{
i2c_start();            /* Start condition               */
i2c_write(0x58);        /* Send out Slave address; Write */
i2c_write(0);           /* Send out Command 1            */
i2c_write(channel_0);   /* Send out datum to channel 0   */
i2c_write(0x01);        /* Send out Command 2            */
i2c_write(channel_1);   /* Send out datum to Channel 1   */
/* Updates both channels                                 */
i2c_stop();             /* Stop condition                */
}
```

`i2c_write(value);`
Sends a single byte over the bus.

`#use i2c(master, scl=PIN_A0, sda=PIN_A1, fast)`
This is a directive by which the programmer informs the compiler which pins are used for the I^2C lines, the fast or standard protocols and Master or Slave mode. The SSP hardware can be designated for the latter situation.

The key characteristic of the various serial protocols discussed up to now is that a clock signal is transmitted by the Master, which allows the Slave to receive or transmit data in perfect synchronization. An alternative approach is to send data under the assumption that the transmitter and receivers are running at approximately the same frequency. This **asychronous** protocol has been in use for data communications system for over a century to send alphanumeric data over telegraph, telephone and radio links to implement the Telex system.

One of the features of early computer development in the 1940/1950s was the extensive use of existing technology. An essential adjunct of any computer-oriented installation is a data terminal. At that time the communications industry made considerable use of the teletypewriter (TTY).[9] Serial data were converted between serial and parallel formats in the terminal itself as well as providing keyboarding and printing functions.

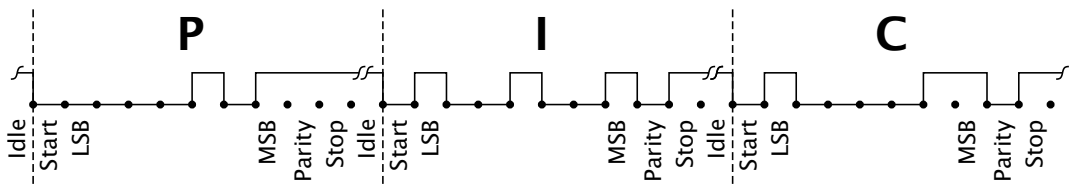


Fig. 12.18 Transmitting the message string "PIC" in the asynchronous serial mode, with odd one's parity and a minimum of one stop bit.

Until the early 1980s, TTYs were electromechanical machines, driven by a synchronous electric motor. This meant that synchronization between remote terminals could only be guaranteed for short periods. To get around this problem, each word transmitted was proceeded by one Start bit and followed by one or more Stop bits. A typical example is shown in Fig. 12.18. While the line is idling, a logic 1 (break level) is transmitted. A logic 0 signals the start of a word. After the word has been sent, a logic 1 terminates the sequence. Electro-mechanical terminals typically print ten characters per second, and require a minimum of

[9] Literally a 'typewriter from afar'; Greek, tele = far.

two Stop bits. This requires a transmission rate of 110 bits per second, or 110 **baud**.[10]

The first purely electronic terminals required only one Stop bit, and could print at 30 characters per second, giving a rate of 300 baud. Traditionally communication channels use multiples of 300; eg. 1200, 2400, 4800, 9600…. PC serial ports can run up to 19,200 baud. However, this $\times 300$ rate is not necessary as long as receiver and transmitter are running at the same nominal rate.

Typically a receiver on detecting an incoming datum will try and sample each bit at approximately mid point. This means that a frequency drift of ± 0.5 bit time can be tolerated in the space of ten bits. Thus the receiver and transmitter local clocks must be within $\pm 5\%$. The two will be resynchronized at the start of each datum.

Although not the most efficient of techniques, the asynchronous protocol outlined here has the major advantage of being an international standard. There are several variants; for instance the word can typically be from five to nine bits long. In our example the word length is eight bits with the eighth bit being used to provide a limited error checking capability. This **parity bit** is set in our example so that the number of 1s in the word is always odd. This can be checked at the receiver (see SAQ 3.9 on page 73) to detect a single bit error.

The original teleprinter code developed by Emile Baudot in 1875 is only five bits long.[11] Here the string "PIC" is coded as 10110 00110 01110. Although limited in capability, its key advantage over Morse code (Samual Morse, 1840) was its fixed length (compare with ·−−·−·−·−·) which considerably simplifies the design of the transmitter and receiver. However, Morse code is more efficient as the number of bits is approximately inversely proportionally to a letter's statistical frequency of use.

The 7-bit ASCII code of Table 1.1 on page 5, first adopted in 1963, was the first code specifically developed for computer communication systems. In 8-bit systems the extra 128 code patterns is usually utilized to add a selection of accented, mathematical and graphic symbols rather than for parity. However, parity can be accommodated by using a 9-bit word format.

For our example we have adopted a format of one Start, eight data with no parity and one Stop bit. Using a bit banging approach, as we have already done for our SPI and I^2C protocols, is straightforward provided

[10] Strictly the baud rate is a measure of information rate. For a simple baseband system this is equal to the bit rate. However, this equality is not always true. For example, a telephone modem can use a di-bit modulation scheme where groups of bits two at a time give a carrier tone phase shift of 0°, 90°, 180° and 270° phase shift for the patterns 00, 01, 10, 11 respectively. In this case the baud rate is four times the bit rate.

[11] Actually the first binary coded alphanumeric code was devised by Francis Bacon in around 1600. It too was a 5-bit code.

that we have an accurate $\frac{1}{2}$-bit delay. For example, for a 4800 baud link this would be 104 μs. As the delay is so short we can use an in-line approach using a macro in the same manner as in Program 12.6 rather than the subroutine approach of Program 6.8 on page 159.

Program 12.10 A baud-rate delay macro showing a half 9600 baud period delay at 20 MHz evocation.

```
              include "p16F84a.inc"
              #define XTAL  d'20'
              #define BAUD  d'9600'
              #define N    (XTAL*d'980000')/(8*BAUD)

Baud_delay macro
              local   BAUD_LOOP
              if(XTAL>d'12')
               movlw  N/9      ; The delay parameter
BAUD_LOOP      addlw  -1       ; Decrement
               nop
               nop
               nop
               nop
               nop
               btfss  STATUS,Z ; Until zero
                goto  BAUD_LOOP
              endif

              if((XTAL>=6)&&(XTAL<=d'12'))
               movlw  N/5      ; The delay parameter
BAUD_LOOP      addlw  -1       ; Decrement
               nop
               btfss  STATUS,Z ; Until zero
                goto  BAUD_LOOP
               endif

              if(XTAL<6)
               movlw  N/4      ; The delay parameter
BAUD_LOOP      addlw  -1       ; Decrement
               btfss  STATUS,Z ; Until zero
                goto  BAUD_LOOP
               endif
               endm
```

The macro shown in Program 12.10 is designed to give a suitable $\frac{1}{2}$-bit delay for a range of baud rates from 1200 through 9600 and crystal frequencies of 4 through 20 MHz. Both BAUD and XTAL constants are defined in the program head by the programmer; the example given in the listing showing a baud rate of 9600 and crystal frequency of 20 MHz.

The kernel of our macro is the decrement loop:

```
          movlw  K        ; 1~          ; K is the delay const
BAUD_LOOP addlw  -1       ; K~          ; Decrement
          btfss  STATUS,Z ; ((K-1)+2)~  ; until zero
           goto  BAUD_LOOP; 2(K-1)~
```

which gives a total of $4K$ cycles delay, where each cycle is 4/XTAL microseconds. This can be increased by padding with nop instructions, each adding K cycles to the total.

For any given baud rate we require $\frac{10^6}{2\times \mathrm{BAUD}}$ microseconds for a $\frac{1}{2}$ bit period delay; so to evaluate the value of K we need to calculate the total number N of 4/XTAL cycles.

$$\begin{aligned} \mathrm{N} \times \frac{4}{\mathrm{XTAL}} &= \frac{10^6}{2 \times \mathrm{BAUD}} \\ \mathrm{N} &= \frac{10^6 \times \mathrm{XTAL}}{8 \times \mathrm{BAUD}} \end{aligned}$$

In the macro of Program 12.10 N has been defined accordingly. To determine the constant to be loaded into W at the beginning of the loop this value is divided by the total delay cycles in the loop. For example, if XTAL is greater than 12 MHz then the five extra nop instructions bring the total delay to nine cycles, hence the initial constant K is N/9. Actually the value of K is can be reduced by around 2% to compensate for the instructions outside the macro; hence the use of 980,000 in Program 12.10 in the definition of N rather than 1,000,000 (10^6).

Notice the use of the local directive to qualify a label inside a macro; in this case BAUD_LOOP. This ensures that when the macro is used several times, the assembler will not object to the same named label appearing more than once in the one program.

With our delay macro in situ, the basic input/output subroutines of Program 12.11 are similar to our bit banging SPI subroutines. The PUTCHAR subroutine simply brings the TX pin low for two Baud_delay periods and then toggles the pin eight times mirroring the data in DATA_OUT least-significant bit first – the opposite order to SPI/I^{2}C. Finally TX is held high for the same period to give the Stop/Idle condition.

The input GETCHAR counterpart is more complex. After an Idle state a low-going voltage at pin RX will be treated as a Start bit. However, if the data stream is subsequentially sampled at intervals of one bit period (two evocations of Baud_delay) then as this is just at the transition point of the transmitter, any drift in the two clock rates may cause errors. To avoid this, a half bit period is evoked and then the state of RX checked to ensure that the Start bit is still present. If it is, then subsequent samples are taken at two Baud_delay periods, which is approximately at the bit center point. Better noise rejection could be obtained by sampling at a

Program 12.11 Asynchronous formatted input and output subroutines.

```
; ************************************************************
; * FUNCTION: Transmits one 8-bit byte in asynchronous format *
; * FUNCTION: Baud rate can be 1200 - 9600 for XTAL 1 -- 20MHz*
; * RESOURCE: Macro Baud_delay giving a 0.5 bit delay; COUNT  *
; * ENTRY   : 8-bit datum in DATA_OUT, XTAL & BAUD predefined *
; * EXIT    : Contents of DATA_OUT 00h, byte TXed             *
; ************************************************************
PUTCHAR      movlw  8            ; Eight data bits
             movwf  COUNT
             bcf    PORTA,TX     ; Start bit
             Baud_delay          ; 2x0.5 bit delay
             Baud_delay
; Now shift out data, LSB first
PUTCHAR_LOOP rrf    DATA_OUT,f   ; Rotate right into Carry
             btfss  STATUS,C     ; Test Carry bit
              goto  ITS_A_0      ; IF 0 THEN output a 0
             bsf    PORTA,TX     ; ELSE output a 1
             goto   PUTCHAR_NEXT ; and continue
ITS_A_0      bcf    PORTA,TX     ; Output a 0
PUTCHAR_NEXT
             Baud_delay          ; One-bit duration
             Baud_delay
             decfsz COUNT,f      ; Repeat eight times
              goto  PUTCHAR_LOOP
             bsf    PORTA,TX     ; Stop bit
             Baud_delay
             Baud_delay
             return
; ************************************************************
; * FUNCTION: Receives one 8-bit byte in asynchronous format  *
; * FUNCTION: Baud rate can be 1200 - 9600 for XTAL 1 -- 20MHz*
; * RESOURCE: Macro BAUD_DELAY giving a 0.5 bit delay; COUNT  *
; * ENTRY   : XTAL & BAUD predefined                          *
; * EXIT    : DATA_IN holds the received byte.                *
; * EXIT    : Err is 00 if no Framing error ELSE -1           *
; ************************************************************
GETCHAR      movlw  8            ; Eight data bits
             movwf  COUNT
             clrf   ERR          ; Zero Error byte
GETCHAR_START
             btfsc  PORTA,RX     ; Poll for 0
              goto  GETCHAR_START
             Baud_delay          ; Hang around for 0.5 bit time
             btfsc  PORTA,RX     ; Check; is it still low?
              goto  GETCHAR_START
             Baud_delay          ; IF yes THEN hang around
             Baud_delay
GETCHAR_LOOP bcf    STATUS,C     ; Clear Carry
             rrf    DATA_IN,f    ; Shift 0 into datum
             btfsc  PORTA,RX     ; Check; is input high?
              bsf   DATA_IN,7    ; IF yes THEN set bit in datum
             Baud_delay
             Baud_delay
             decfsz COUNT,f      ; Do eight times
              goto  GETCHAR_LOOP
             btfss  PORTA,RX     ; Look for a Stop bit (High)
              decf  ERR,f        ; IF low THEN signal an error
             return
```

higher rate and then taking a majority decision regarding the logic state of the incoming voltage.

After the eight data bits have been shifted into DATA_IN, the Stop bit is checked for 1. If Stop is 0 then a **Framing error** has occurred. This is signalled by returning a value of −1 in ERR. Other more elaborate schemes may return a variety of error types. For example, where parity is used then a Parity error can be returned.

As an example, if we wish to transmit the three characters PIC then the following code fragment would implement our task. For convenience the assembler allows the programmer to represent ASCII codes in delimited single quotes to represent their ASCII equivalent, as described on page 223.

```
movlw  'P'      ; ASCII for P is 50h
movwf  DATA_OUT ; Put in store
call   PUTCHAR  ; Send it out
movlw  'I'      ; ASCII for I is 49h
movwf  DATA_OUT ; Put in store
call   PUTCHAR  ; Send it out
movlw  'C'      ; ASCII for C is 43h
movwf  DATA_OUT ; Put in store
call   PUTCHAR  ; Send it out
```

Handling serial communications this way is only really satisfactory for very simple situations. For example, if the RX pin is not continually monitored a transmission can be missed or synchronization lost. Also it is difficult to implement a full-duplex link. In addition the procedure is software intensive with most of the processing power being wasted in delay loops. The situation can be improved somewhat by using an internal timer to generate the baud delay and by using interrupt-driven techniques. However, the majority of 28+pin PICs have an integral communications port to automatically deal with asynchronous transmission.

One of the first applications of the then new LSI fabrication techniques in the late 1960s, was the implementation of a dedicated hardware asynchronous serial port known as the **Universal Asynchronous Receiver Transmitter**. The **UART**[12] was already in production by the time microprocessors were developed. Most PCs, even in the 1970s, had a serial port implemented by a UART, as do current systems. As well as dealing with shifting, error checking and interrupt handling, most UARTs also have an integral baud-rate generator which can be set up in software to give the correct bit frequency.

Figure 12.19 shows a simplified model of the basic PIC **USART – Universal Synchronous-Asynchronous Receiver Transmitter** as the port has a synchronous mode (SYNC = 1) which will not be discussed here. The core of the USART is the Transmit and Receive registers and their

[12] Sometimes known as the Asynchronous Communication Interface Adapter or ACIA.

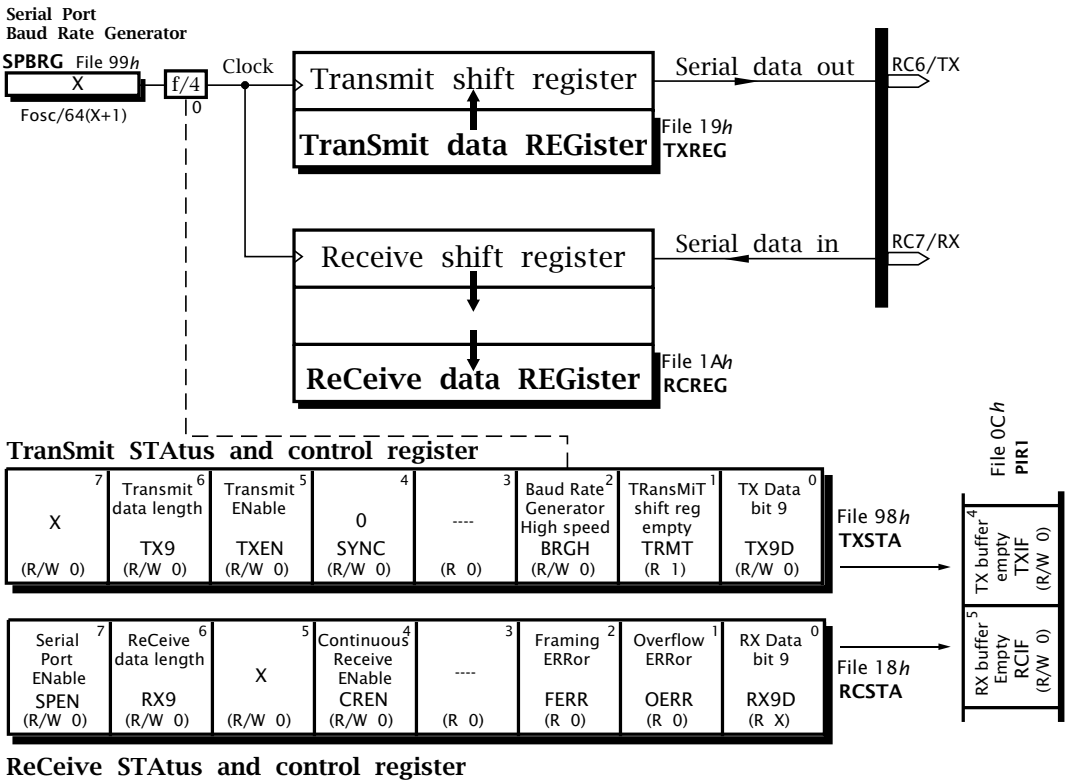


Fig. 12.19 The PIC USART configured for asynchronous communication.

associated buffers and Status registers. To enable the overall USART the **Serial Port ENable** (**SPEN**) bit in the **ReCeive STAtus register** (RCSTA[7]) at File 18*h* must be set.

Transmission

The transmitter logic is enabled when the **TranSmit ENable** (**TXEN**) bit in the **TranSmit STAtus register** (TXSTA[5]) at File 98*h* is set. To send a character the datum must be moved to the **TranSmit data REGister** (**TXREG**) at File 19*h*, whence it will be transferred to the **Transmit shift register** and shifted out of pin TX, which is shared with RC6. If a 9-bit format is required the TX9 bit in TXSTA[6] must be set to 1 and the ninth bit placed in bit 0 of the same register *before* moving the lower eight bits into TXREG. If the Transmit shift register is not empty; that is it is in the process of shifting out a previous datum, then the new datum will remain in the TXREG buffer register awaiting the completion of transmission before being transferred.

Bit 1 of the TranSmit STAtus register reflects the state of the Transmit shift register whilst the **TranSmit Interrupt Flag** (**TXIF**) in bit 4 of the Peripheral Interrupt Register 1 (PIR1) is automatically set when the TXREG buffer is empty and ready for reloading. If an interrupt on TX buffer is empty is required, the corresponding TXIE mask bit in the Peripheral

Enable Register 1 (PIE1[4]) must be set – see Fig. 14.10 on page 408. TXIF is automatically cleared whenever a datum is written into the TXREG.

Reception

Once a Start bit is detected at pin RX then the succeeding eight or nine bits are shifted into the 2-deep **ReCeive data REGister** (**RCREG**) at File 1A*h* pipeline irrespective of what is going on at the transmitter section. If a 9-bit receive protocol has been selected with RX9 set to 1 in RCSTA[6] then the ninth bit can be read in the RX9D bit of the same status register.

When a datum has been received, it is automatically stored in the top RCREG buffer whence it moves to the lower buffer, provided that no datum is still waiting to be read. **ReCeiver Interrupt Flag** (**RCIF**) is automatically set whenever a datum is waiting for collection and this can be used to generate an interrupt if the RCIE mask bit is set; as well as the GIE and PEIE global masks. RCIF is cleared whenever a datum is read. If a datum is waiting in the top buffer, then RCIF is immediately set again showing that there is another datum ready for collection.

If a third character has been received and the 2-deep receive pipeline is full then the **Overflow ERRor** (**OERR**) bit at RCSTA[1] will be set and this newly received datum will be lost. The RCREG can still be read twice to retrieve the two buffered bytes. However, to clear OERR the receive logic must be reset by clearing the **Continuous Receive ENable** (**CREN**) bit in RCSTA[4] and then setting it again.

The **Framing ERRor** (**FERR**) bit in RCSTA[2] will be updated by reading the RCREG on the next received datum. Both FERR and any ninth received bit are double buffered in the same way as the received data and so should be read/checked first *before* the main datum is read as this will empty the pipeline and therefore change these auxiliary bits.

Serial Port Baud-Rate Generator, SPBRG

This is basically a programmable 8-bit counter followed by a switchable frequency ÷4 flip flop chain which can be set up to give the appropriate sampling and shifting rates for the desired baud rate, based on the PIC's crystal frequency XTAL giving:

$$\text{Baud rate} \quad \frac{\text{XTAL}}{64\times(X+1)} \quad \text{Low-speed}$$

$$\text{Baud rate} \quad \frac{\text{XTAL}}{16\times(X+1)} \quad \text{High-speed}$$

where X is an 8-bit datum written into SPBRG at File 99*h* who's value is $\frac{\text{XTAL}\times 10^6}{64\times\text{BAUD}} - 1$. For example, if we require a baud rate of 4800 on a 20 MHz device, then a value X = 64 will give a baud rate of 4808, an error of +0.161%. At 20 MHz the maximum baud rate is 312,500 whilst the lowest rate is 1221. A baud rate of 1.25 Mbaud is obtainable at 20 MHz in the high-speed mode with SPBRG = 1, but Microchip do not advise this

mode for older devices, such as the PIC16C74/74A,[13] as receive errors can occur.

Actually, the SPBRG produces higher frequencies than the base baud rate, to enable the USART to take three samples around bit midpoints and adopt a majority decision. In the low-speed mode a sampling rate of $\times 16$ is used, as is the case for newer devices in the high-speed mode. Older devices use $\times 4$ for the high-speed mode.

To illustrate how to use the USART we will repeat our GETCHAR and PUTCHAR subroutines using hardware. Firstly, in the main program we have to set up the Serial Port Baud Rate Generator and both Transmit and Receive Status/Control registers. Assuming, as in Program 12.10, the programmer has defined the constants XTAL and BAUD then we can let the assembler evaluate the arithmetic to give us the value of X to put in the SPBRG. With this in mind, the initialization code would look something like:

```
        include "p16c74b.inc"

        #define  BAUD  d'4800' ; For example 4800 baud rate
        #define  XTAL  d'8'    ; 8MHz crystal
        #define  X     ((XTAL*d'1000000')/(d'64'*BAUD))-1

START   bsf      STATUS,RP0    ; Change to Bank1
        bsf      TRISC,7       ; RX pin is set as an input
        bcf      TRISC,6       ; TX is set as an output
        movlw    X             ; Move X to Baud Rate Generator
        movwf    SPBRG
        movlw    b'00100000'   ; 8 data bits, TX enabled
        movwf    TXSTA         ; Low speed SPBRG mode
        bcf      STATUS,RP0    ; Back again to Bank0
        movlw    b'10010000'   ; USART enabled, 8 data bits
        movwf    RCSTA         ; Receiver enabled
```

With the USART enabled the subroutines are coded in Program 12.12. PUTCHAR is simply a matter of polling TXIF waiting for it to go to 1 and then copying the datum to the TranSmitter REGister.

The input GETCHAR is a little more complex if some error checking is to be incorporated. The subroutine polls the state of RCIF which goes to 1 whenever there is data to be read. Also returned is the variable ERR which is $00h$ if there is no problem, -1 if a Framing error occurred, -2 if a Overflow situation is sensed and -3 if both errors occurred. In these latter situations OERR is zeroed by resetting the receiver logic. After the error conditions have been checked the data is read from the ReCeive REGister. This is done after checking to avoid altering the appropriate error flags.

[13] For example, the receive logic sampling rate of the PIC16C74B has been altered to eliminate these errors.

Program 12.12 The USART-based I/O subroutines.

```
; ************************************************************
; * FUNCTION: Transmits one 8-bit byte in asynchronous format *
; * RESOURCE: PIC USART                                       *
; * ENTRY   : 8-bit datum in DATA_OUT                         *
; * EXIT    : Contents of DATA_OUT unchanged, byte TXed       *
; ************************************************************
PUTCHAR btfss    PIR1,TXIF     ; Check, is TX buffer full?
         goto    PUTCHAR       ; IF not THEN try again
        movf     DATA_OUT,w    ; ELSE get datum
        movwf    TXREG         ; and copy to USART TX register
        return

; ************************************************************
; * FUNCTION: Receives one 8-bit byte in asynchronous format  *
; * RESOURCE: PIC USART                                       *
; * ENTRY   : None                                            *
; * EXIT    : DATA_IN holds the received byte.                *
; * EXIT    : ERR is 00 if no error. Framing ERRor only = -1  *
; * EXIT    : ERR = -2 if Overflow ERRor and -3 if both types *
; ************************************************************
GETCHAR clrf     ERR           ; Zero flag byte
        btfss    PIR1,RCIF     ; Check, is there a char ready?
         goto    GETCHAR       ; IF not THEN try again
; Error return
        btfss    RCSTA,FERR    ; Was there a Framing error?
         goto    CHECK_OERR    ; IF not THEN check for Overflow
        movlw    -1            ; ELSE record a Framing error

CHECK_OERR
        btfsc    RCSTA,OERR    ; Check for Overflow ERRor
         goto    GET_EXIT      ; IF none THEN complete
        decf     ERR,f         ; Otherwise register error
        decf     ERR,f
        bcf      RCSTA,CREN    ; and reset the logic
        bsf      RCSTA,CREN
GET_EXIT
        movf     RCREG,w       ; Get datum
        movwf    DATA_IN       ; and put away
        return
```

Some systems may not wish the processor to hang up waiting for a character which is a long time in coming. In such cases an alternative input subroutine, perhaps called `getch`, could return an `ERR` of -1 if the return was empty handed. Another approach would be to generate an interrupt each time an incoming character is sensed rather than using a polling technique.

In the case of the CCS **C** compiler the `#use rs232` directive tells the compiler which pins are to be used for `RX` and `TX`. The normal **C** I/O functions, such as `printf()`, use these pins as their link to the standard

channel. If these pins are specified as `PIN_C6` and `PIN_C7` then where the part has a built-in USART this will be used instead of a software technique.

There is more to setting up a communication link than establishing a suitable protocol. PIC devices have normal logic voltage and current levels which are not intended for connections greater than 30 cm (1′). Although with care[14] distances considerably in excess of this can be employed, in situations with relatively fast bit rates different signalling techniques have to be used.

In the era of electromechanical TTYs the de facto 20 mA loop standard was in common use. This uses zero and 20 mA current to signal logic `0` and logic `1` respectively. Use of current means that line attenuation is not a problem (as current out must equal current in) and this level of current was sufficient to directly activate the receiver solenoid relay.

Current sources are realized by using high voltages in series with a large resistance. The latter gives long time constants, which, while adequate in the area of 110 baud rates, did not transfer well to the introduction of electronic terminals, UARTs and modems. **RS-232**[15] was introduced in 1969 as the standard interface for connecting an item of Data Terminal Equipment (DTE), such as a terminal, to approved Data Circuit terminating Equipment (DCE), typically a modem. Thus, not only did it define signalling levels, as shown in Fig. 12.20(a), but also various control and handshake lines, some of which are shown in Figs. 12.20(d) and 12.21. For example the modem would signal back to the DTE that a telephone link had been opened with the remote DTE by activating the Clear To Send (CTS) handshake signal. Two data lines plus a ground line are needed for a full duplex transmission circuit.

The RS-232 standard has a range of 15 m (50′) at a maximum rate of 20 kbaud, which it achieves by mapping logic `0` (often called a **space**) to typically +12 V and logic `1` (often called a **mark**) to typically −12 V. The receiver can distinguish levels down to ±5 V. The **RS-423 standard** (1978) in Fig. 12.20(b) is similar but can manage 1.2 km (6000′) at up to 80 kbaud and 10 Mbaud at 12 m (40′) with up to ten receivers.

Both RS-232 and RS-423 are **unbalanced** (or single-ended) standards, where the receiver measures the potential between signal line and ground reference. Even though the transmitter and receiver grounds are usually connected through the transmission line return, the impedance over a long distance may support a significant difference in the two ground potentials, which will degrade noise immunity. Furthermore, any noise induced from outside will affect signal lines differently from the ground return due to their dissimilar electrical characteristics – hence the name **unbalanced**.

[14] Or sometimes ignorance!

[15] Defined in USA as the Electronics Industries Association EIA 232-E standard and as the V24 interface by the CCITT in Europe.

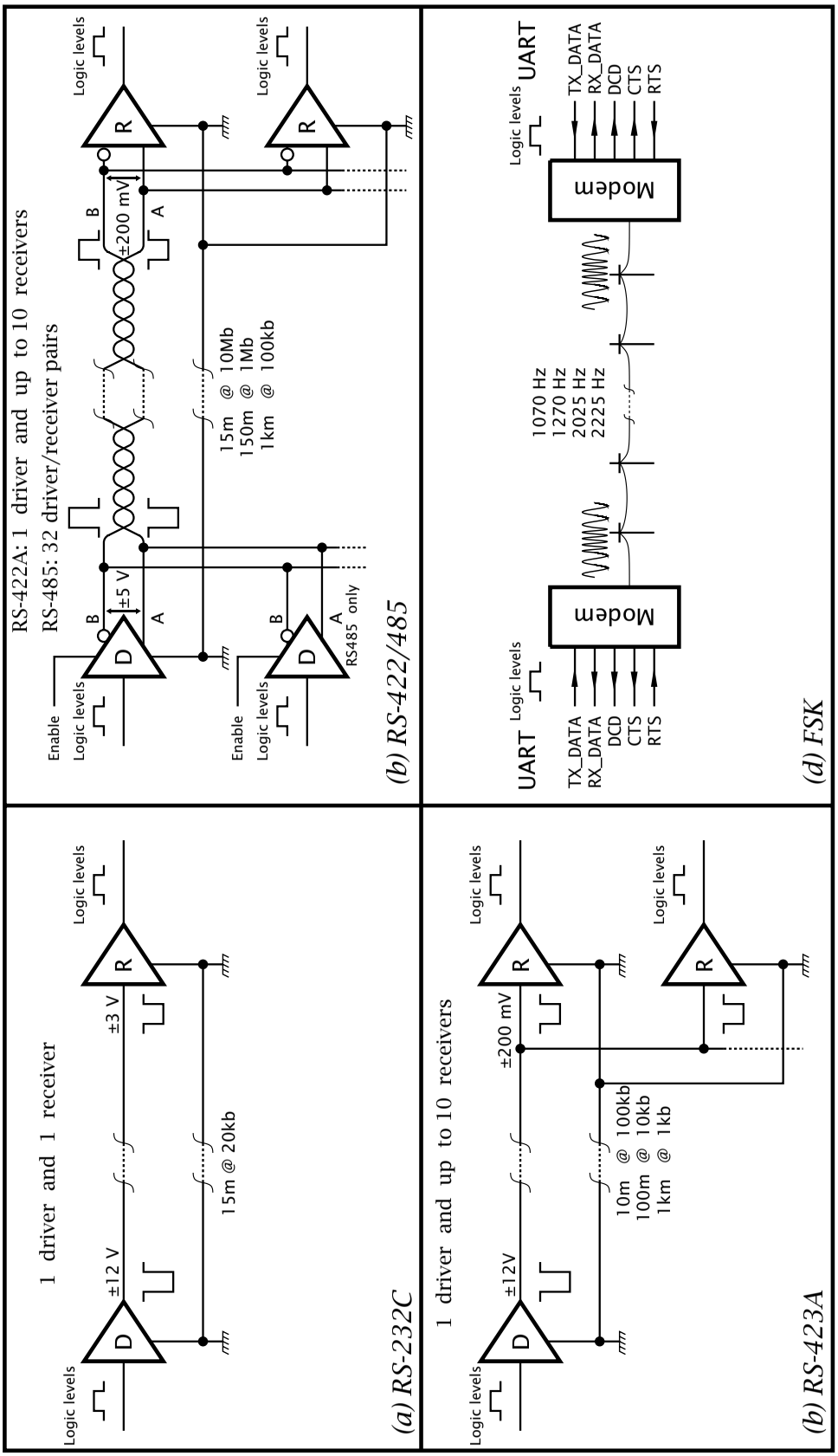


Fig. 12.20 Some signalling configurations.

The **RS-422** (1978) and **RS-485** (1983) standards are described as **balanced**. Here each signal link comprises *two* conductors, normally twisted around each other, known as **twisted pair**. The logic level is represented as the *difference* of potential across the conductors, not the difference from ground. Calling the conductors A and B, then logic 0 is represented as A<B and logic 1 by A>B. A difference of more than ±200 mV at the receiver is sufficient to establish the logic level and the transmitter will typically generate a $\Delta V = \pm 5$ V. As the A and B conductors have the same characteristics and are tightly wound together they represent similar targets for induced noise. As the same noise voltage appears in *both* conductors and the receiver only distinguishes *differences*, rejecting common-mode voltages up to ±7 V, then the noise immunity of these balanced links is clearly superior to unbalanced schemes. Commercial twisted-pair cables, used in Local Area Networks (LANs), often carry three or four pairs of conductors, each link having a different twist pitch to reduce induction between links.

The main difference between the RS-422 and RS-485 standards is the provision in the latter case for multiple transmitters as well as receivers to implement multi-drop LANs. As only one transmitter can be active at any one time, an RS-485 transmitter buffer must have an enable input, to select the master device. The single RS-422 transmitter has no need to be disabled.

RS-232 was originally designed for DTE-modem interconnection, although its use is now much more varied – see Fig. 12.21. Figure 12.20(d) shows a simple Frequency Shift Keying (FSK) full duplex system with the mark/space of one channel being represented by the tones 1070/1270 Hz and the other by 2025/2225 Hz; frequencies which fit well inside the normal telephone link bandwidth of 300 – 3400 Hz. Handshake lines DCD (Data Carrier Detect), CTS (Clear TO Send) and RTS (Ready To Send) are used to control the sequence of operations prior to and terminating the communication of data.

Many modem schemes currently use Phase Shift Keying (PSK) where typically at least eight different phases in 45° steps of a single tone are used to encode 3-binary bit code groups (tri-bits) in any one time slot. In this way the baud rate may be increased with the same signalling rate, albeit at the expense of noise immunity, as witnessed by the steady increase in PC-based home telephone internet data rates in recent years up to 56 kbaud.

As an example, Fig. 12.21 shows the connection between a PIC and the serial port of a PC – or any device with an asynchronous RS-232 port. The Maxim MAX233 dual RS-232 transceiver translates from +12 V to 0 V (logic 0) and −12 V to +5 V (logic 1). If handshake lines are not being used, as is usual in simple links, the PC can be 'fooled' into treating the interface as ready to accept data by linking as shown in the diagram. For

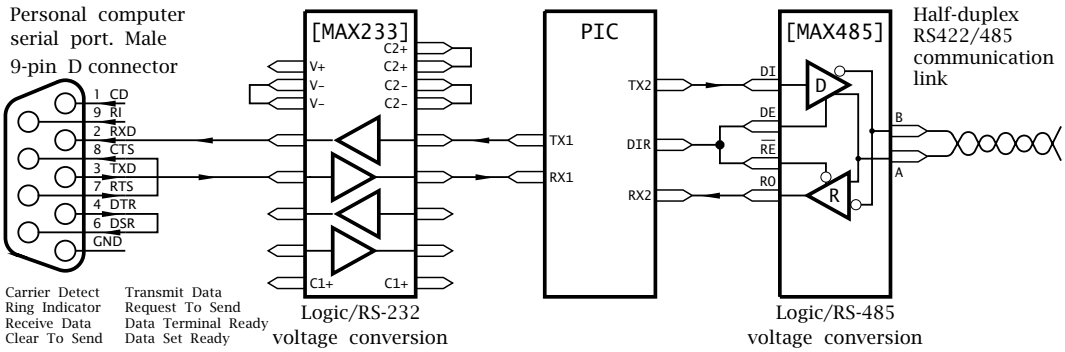


Fig. 12.21 Communicating with a PC via an RS-232 link and the outside world.

instance, the serial port UART's RTS is looped back to CTS. The MAX233 has two transmit and two receive buffers in all and thus can be used to buffer some additional handshake lines if required.

In Fig. 12.21 the same PIC is shown driving a half duplex RS-485 link using a Maxim MAX485 voltage converter. Each buffer has a separate Enable of the opposite logic polarity. The PIC can activate the appropriate buffer depending on the communication direction. Alternatively the MAX485 can be used to implement a full duplex channel using two separate links.

The RS-485 link need not use the asynchronous protocol. Any suitable synchronous protocol can be buffered to RS-485, but of course a separate clock channel will be needed.

Examples

Example 12.1

In Example 11.2 we designed a subroutine to compare a *fixed* number `TRIP` with the byte read in from Port B. In some cases it may be necessary to have the software adapt to changing circumstances, altering the trigger value by reading updates from outside. Rather than using up another eight port lines it is proposed that the update be fed in from an outside agency in series at pin `RA4`, with `RA3` being used as the clock line. With the assumption that each data bit is set up when the Clock line is low write a subroutine to read in a new value into memory at `TRIP`.

Solution

One solution is shown in Program 12.13. The Clock line is monitored for high, during which time the Data is stable. By mirroring the state of the Data line into the Carry flag the datum is rotated bit by bit into memory.

After each shift the loop is not completed until the Clock line again goes low.

This is similar to subroutine SPI_READ in Program 12.2 except that the clock is generated from outside; that is the PIC is acting as a Slave. This causes problems in a real system where the PIC Slave must be able to tell the Master when it wants a new byte. This could be done by using another port line as a Clear To Send handshake which will interrupt the Master and initiate the conversion. Of course the Master could be another PIC and if so we have an economical way of connecting two PICs together. If PICs with integral serial ports are used then interrupts can be automatically generated and this is a frequently used way of implementing multi-processor networks.

Program 12.13 Updating Program 11.4's trip value.

```
; ***********************************************************
; FUNCTION: Shifts in value for TRIP which is subsequently  *
; FUNCTION: used as one operand for subroutine COMP          *
; ENTRY   : Data bit changes at RA4 when at RA3 is low       *
; EXIT    : COUNT is 00, datum is in TRIP                    *
; ***********************************************************
SER_TRIP  movlw  8            ; Bit loop count
          movwf  COUNT
SER_TRIP_LOOP
          btfss  PORTA,3      ; Wait for Clock to go high
           goto  SER_TRIP_LOOP
          bcf    STATUS,C     ; Carry = 0
          btfsc  PORTA,4      ; Is Data line high?
           bsf   STATUS,C     ; IF yes THEN Carry = 1
          rlf    TRIP,f       ; Shift bit in
SER_TRIP_LOOP2
          btfsc  PORTA,3      ; Wait for Clock to go low
           goto  SER_TRIP_LOOP2
          decfsz COUNT,f
           goto  SER_TRIP_LOOP
          return
```

Example 12.2

Design and code the I2C_IN counterpart of the I2C_OUT subroutine of Program 12.7. You may assume that the same variables are available and that the output datum is in DATA_IN on entry.

Solution

The I2C_IN subroutine of Program 12.14 shifts the datum in file register DATA_IN through the Carry flag eight times with pin SDA mirroring this bit state. At the same time the Clock line SCL is toggled in according

Program 12.14 Reading in a byte using the **I²C** protocol.

```
; ************************************************************
; * FUNCTION: Reads in byte from Slave with optional ACK/NACK *
; * ENTRY   : ACKNO = 00 for ACK ELSE NACK                    *
; * RESOURCE: START and STOP subroutines, Delay_600 macro     *
; * EXIT    : DATA_IN holds datum sent from slave             *
; * EXIT    : ACK or NACK sent to Slave, SCL low              *
; ************************************************************
I2C_IN    bcf    INDF,SCL     ; Make sure that Clock line is low
          movlw  8            ; Loop count = 8
          movwf  COUNT

I2C_IN_LOOP
          bcf    INDF,SCL     ; Clock low
          Delay_600           ; For minimum period
          Delay_600
          bsf    INDF,SCL     ; Clock high
          bcf    STATUS,C     ; Carry = 0
          btfsc  INDF,SDA     ; Check state of incoming bit?
           bsf   STATUS,C     ; IF 1 THEN make Carry = 1
          rlf    DATA_IN,f    ; and rotate it into the datum
          decfsz COUNT,f      ; Decrement loop count
           goto  I2C_IN_LOOP  ; and repeat eight times

; Now determine if Acknowledge is to be sent
           bcf   INDF,SCL     ; Clock low
           bsf   INDF,SDA     ; Data output float (NACK)
           movf  ACKNO,f      ; Test the caller's wish
           btfsc STATUS,Z     ; IF non zero THEN leave as NACK
            bcf  INDF,SDA     ; ELSE bring low to signal ACK
           Delay_600          ; Keep Clock low
           Delay_600
           bsf   INDF,SCL     ; Now high
           Delay_600
           bcf   INDF,SCL     ; Leave with Clock low
           return
```

to the I²C time and protocol specification as in our I2C_OUT subroutine of Program 12.7. In this protocol the Master signals back to the Slave to stop sending data by letting the SDA line float high in the Acknowledge slot in the ninth clock pulse – see Fig. 12.13. The normal low state in this slot is called ACK, whilst the deviant high Acknowledge state is called NACK (No ACKnowledge). To cope with both these situations our I2C_IN optionally generates either situation depending on the state of the variable ACKNO set by the caller. If file register ACKNO is zero on entry then a normal low ACK is sent in this slot. Any non-zero value in this variable causes a high NACK to be sent back to the Slave. The Slave then terminates its transmission and listens for the next Stop/Start condition.

Example 12.3

Many MCU-based products require storage of data in non-volatile memory for retrieval after the system has been powered down. A typical example is the total distance travelled by a car from new, which should be held independently of the state of the car battery. Such data is typically held in Electrically-Erasable Programmable Read-Only Memory (EEPROM) as described on page 28. Many EEPROM devices are available which interface to SPI and I^2C, specifically the I^2C 24LCXXX shown in Fig. 12.22. The 24LCXXX 8-pin serial EEPROMs vary from the 1 kbit 24LC01B to the 256 kbit 24LC256, organized as bytes; i.e. 128 byte to 32 kbyte.

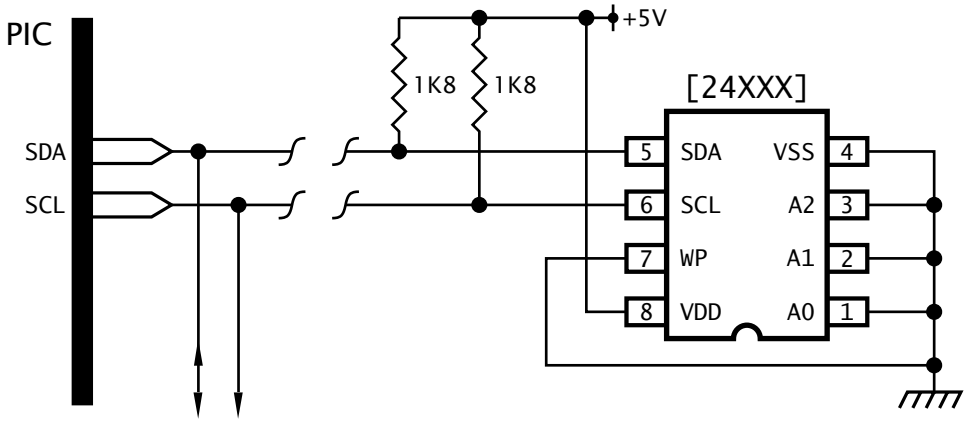


Fig. 12.22 The 24XXX series of I^2C serial EEPROMs.

The 24XXX serial EEPROMs have the following features.

- 400 kHz I^2C compatible (V_{DD} = 5 V), 100 kHz at V_{DD} = 2.5 V.
- Write protection (ROM mode) using the WP pin.
- 2 ms typical Write cycle time.
- 1,000,000 minimum[16] Write cycle endurance per byte cell.
- 3 mA Write, 1 mA Read and 100 μs standby current.
- Internal generation of high programming voltage.

Using a 24LC01B serial EEPROM, show how you could increment a number in the bottom three locations which represents the total distance in either miles or kilometers depending on the market. You may assume that the PIC is interrupted on each mile/kilometer and that this software is part of the interrupt handler. You have the resources of the subroutines of Program 12.7 and 12.14.

Solution

Before writing code to implement our specification we need to look more closely at the protocol used by the 24XXX serial EEPROMs in communi-

[16] 100,000 in the 24LC128 and 24LC256 devices.

cating with the Master PIC. This is encapsulated in the messages shown in Fig. 12.23.

In all cases the Master initiates a data transfer by sending a Start condition followed by a Command byte. The Control byte contains the I^2C Slave address 1010; the chip select address A2 A1 A0 and the R/$\overline{W}$ bit in the order

1	0	1	0	A2	A1	A0	R/$\overline{W}$

. Although the chip select address is shown as part of the Command byte and the three corresponding pins are shown in Fig. 12.22, newer versions of the smaller EEPROMS do not implement this feature. This is because if EEPROM capacity needs to be expanded then it more efficient to replace the device by a pin-identical larger version. For example replacing a 24LC01B by a 24LC08B gives an eightfold increase with no hardware alteration. Larger EEPROMS, such as the 24LC256 do implement chip select address as the method of expansion as additional devices will need to be hung on the bus in this situation. Eight 24LC256s will give a capacity of 256 kbyte of non volatile memory.

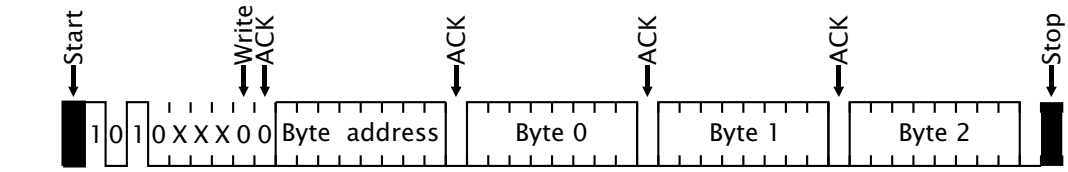
This is normally followed by the address in the EEPROM that data is to be written into or read out of. In the specific case of the 24LC01B the data is arranged as 128 cells, each comprising a byte that can be individually written to or read from. This means that a 7-bit address will fit comfortably in the 8-bit address byte. This scheme will cope with devices up to the 24LC02B but beyond this addresses greater than 8 bit wide are needed. This is done by using the Chip select bits in the Command bit, giving an address width of 11 bits and a capacity of 2 kbytes (16 kbits). For EEPROMs larger than the 24LC16 two Address bytes are used following the Command byte.

The process of sending the byte address to the EEPROM is implemented as a Write action in Fig. 12.23. This is actioned by setting the R/$\overline{W}$ bit low in the command bit. Where a data byte is to be written into the addressed location this byte comes immediately after the Address byte and then followed by a Stop condition. If more than one data byte is transmitted before the Stop then this data is stored in a small on-board buffer and the actual programming will not occur until the Stop condition. The 24LC01B can store eight bytes at a time in a single page, with the *lower three* address bits being incremented on each data byte sent. If this address rolls over, earlier addressed data will be overwritten. The size of this page depends on the device; for example, the 24LC256 uses a 64-byte page. In Fig. 12.23(a) three bytes are shown being written into the 24LC01B. As these locations are to be targeted in the bottom three locations, 00-01-02h, then roll-over will not occur.

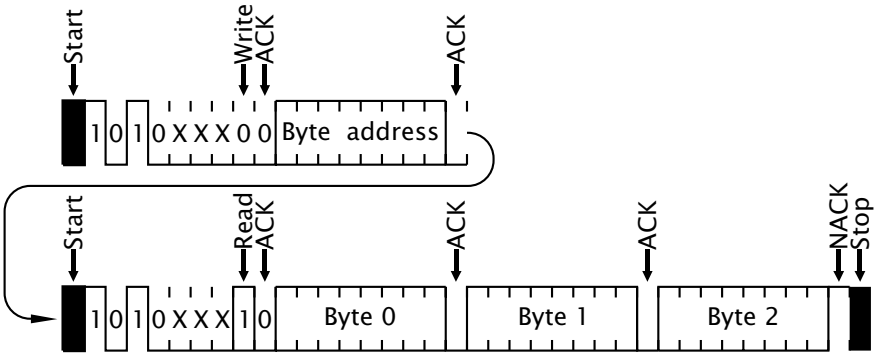
As soon as the Stop condition is received the 24LC01B will commence programming the targeted cells with the buffered data. This process takes typically 2–5 ms across the family. If the Master attempts to initiate a process during this time then the EEPROM will not Acknowledge

Program 12.15 Incrementing the non-volatile odometer count.

```
EXTRA_MILE                  ; Get the three bytes at 00:01:02h
          call    START     ; Start a transmission packet
; Command byte 1 to initialize address
          movlw   b'10100000'; Slave address Master-Write
          movwf   DATA_OUT  ; Copied to pass location
          call    I2C_OUT   ; Send it out
          movf    ERR,f     ; Check for an Acknowledge error
          btfsc   STATUS,Z  ; IF Zero THEN continue
           goto   EXTRA_MILE ; ELSE try again
; Address 00
          clrf    DATA_OUT  ; Pass location
          call    I2C_OUT   ; Send it out

; Command byte 2 to change over to Read
          call    START
          movlw   b'10100001'; Slave address Master-Read
          movwf   DATA_OUT  ; Copied to pass location
          call    I2C_OUT   ; Send it out

; Now read in three bytes
          clrf    ACKNO     ; Enable Acknowledge
          call    I2C_IN    ; Read the High byte in 00h
          movf    DATA_IN,w ; Get byte
          movwf   MSB       ; and put in memory
          call    I2C_IN    ; Read the Middle byte in 01h
          movf    DATA_IN,w ; Get byte
          movwf   NSB       ; and put in memory
          incf    ACKNO,f   ; Signal a NACK
          call    I2C_IN    ; Read the Low byte in 02h
          movf    DATA_IN,w ; Get byte
          movwf   LSB       ; and put in memory
          call    STOP      ; End of Read process

; Now increment 3-byte array
          incf    LSB,f     ; Add one
          btfss   STATUS,Z  ; Is it now zero
           goto   PUT_BACK  ; IF not THEN continue
          incfsz  NSB,f     ; ELSE increment middle byte
           goto   PUT_BACK  ; IF not zero THEN continue
          incf    MSB,f

PUT_BACK  call    START     ; Start the Write process
          movlw   b'10100000'; Write state
          movwf   DATA_OUT
          call    I2C_OUT
          clrf    DATA_OUT  ; Address 00h
          call    I2C_OUT

          movf    MSB,w      ; Get the new High byte
          movwf   DATA_OUT
          call    I2C_OUT
          movf    NSB,w      ; Get the new Middle byte
          movwf   DATA_OUT
          call    I2C_OUT
          movf    LSB,w      ; Get the new Low byte
          movwf   DATA_OUT
          call    I2C_OUT

          call    STOP
```



(a) Writing three bytes.



(b) Reading three bytes.

Fig. 12.23 EEPROM Read and Write waveforms.

following the Start-Control and byte and this can be used as a busy indicator. This polling is shown when the first Control byte is sent out in Program 12.15.

The opposite process of reading bytes from the EEPROM is slightly more involved. As in the previous case a start address has to be written into the device. After this occurs a repeat Start condition is sent with the following Control byte having its R/$\overline{W}$ bit high to indicate Read. The Slave then transmits the byte at the specified location to the Master which Acknowledges receipt and the process continues indefinitely with the address incrementing until the Master does not send an Acknowledge. The Slave then releases the bus and the Master is free to issue a Stop condition. If the initial writing of the start address is omitted then one beyond the last used address is the additional location read from.

The software listed in Program 12.15 follows the process outlined in Fig. 12.23 exactly. Once the initial address 00*h* has been sent the Master goes into a listen mode and three sequential bytes are read from memory terminated by the Master by returning a NACK condition followed by Stop. With the triple-byte distance count in locations `MSB:NSB:LSB` the array is incremented in the usual way. Finally address 00*h* is again written out to the EEPROM followed by the three updated bytes and the process terminated by the Master transmitting Stop.

Example 12.4

It is possible to combine some of the attributes of synchronous I^2C and asynchronous signalling to send data asynchronously in both directions half-duplex along a single link. One example of this is the **1-Wire**[17] interface outlined in Fig. 12.24.

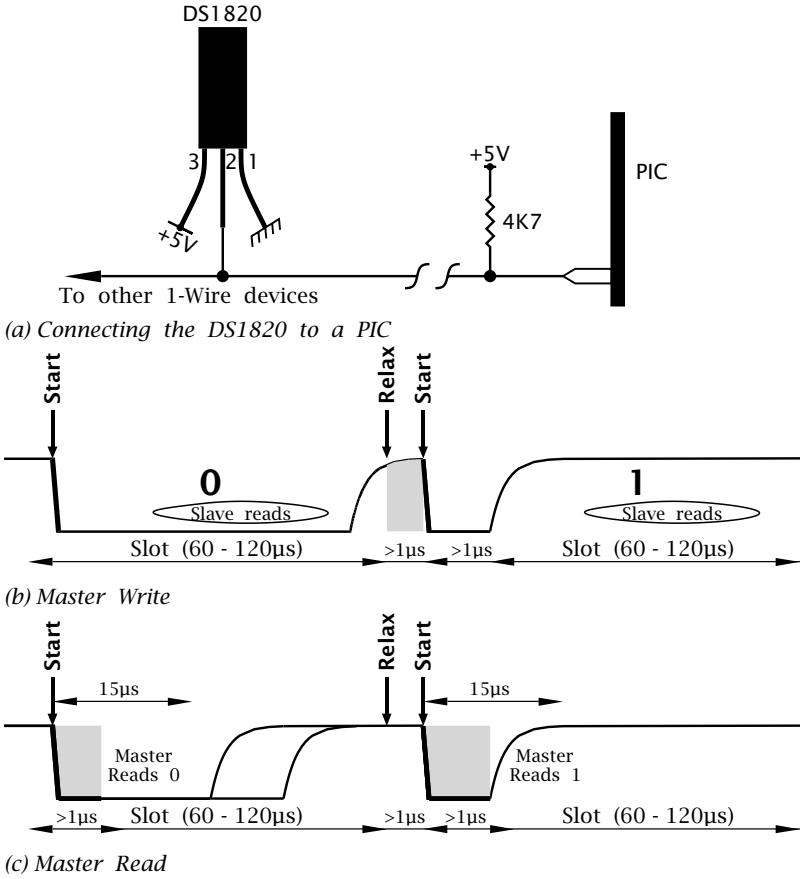


Fig. 12.24 Interfacing the DS1820 1-Wire digital thermometer.

In Fig. 12.24(a) a Dallas Semiconductor DS1820 digital thermometer is shown driven from a single port line with the MCU acting as a 1-Wire Master.

The DS1820 has the following features.

- Measures temperature from −55°C to +125°C in 0.5°C steps as a signed 16-bit datum.
- Converts temperature in 500 ms maximum.
- Zero standby current.
- Can be powered in certain limited circumstances from the data line.
- Multidrop capability.

[17] 1-Wire is a trademark of Dallas Semiconductor.

The various DS1820 functions, such as Convert (44*h*), Read temperature (BE*h*), are initiated by the Master sending the appropriate data as 8-bit codes, each byte comprising a Start condition (‾_) and eight Write slots, as shown in Fig. 12.24(b). As in the I^2C case, the data line DQ is pulled high with a pull-up resistor and the Master simulates the logic 1 state by changing its port line from low to input (see Fig. 12.14(b)). In this state the Master can listen to data sent by the Slave as shown in Fig. 12.24(c).

For our example we are required to write two subroutines that will respectively write a byte to a 1-Wire Slave and read a byte from the Slave.

Solution

From Fig. 12.24(b) we see that writing a bit to a Slave involves the following tasks:

1. The Master starts the process by forcing the data line low for at least 1 μs.
2. The Master either keeps the line low (Write 0) or releases the line (Write 1) for 60 – 120 μs.
3. The Slave reads the line state between 15 – 45 μs later.
4. The Master releases the line (if Write 0) for at least 1 μs to relax the system.

The subroutines of Program 12.16 assume that the port line driving DQ has been set up as described on page 325 for the I^2C bus to give the two states as hard low and open circuit – pulled up high. Also we assume that we have the delay macro Delay_us in situ which gives a $K\mu$s delay, where K is the parameter passed to the macro.

```
Delay_us macro     K       ; K is the number of usec delay
         local     DELAY_US_LOOP
         movlw     (K*XTAL)/(4*3)+1 ; 1~
DELAY_US_LOOP
         addlw     -1       ; Decrement count: N~
         btfss     STATUS,Z ; to zero        : N + 1~
          goto     DELAY_US_LOOP  ;          : 2(N-1)~
         endm
```

Both subroutines begin by driving DQ low for a minimum of 1 μS, defining the Start condition. Writing a single bit to DQ occurs in a slot which has a duration of 60 – 120 μs, and commences with DQ either low or released to be pulled high, defining a Write-0 or Write-1 condition. The Slave samples the state of the data line sometime after 15 μs into the slot. Although the duration of the slot is not critical, care needs to be taken as a low duration of between 480 and 960 μs is interpreted by the Slave as a Reset command (see SAQ 12.3).

Program 12.16 Reading and writing on a 1-Wire system.

```
; ************************************************************
; * FUNCTION: Writes a byte datum to a 1-Wire slave          *
; * RESOURCE: macro Delay_us giving N microsecond delay      *
; * ENTRY   : Datum is in DATA_OUT                           *
; * EXIT    : DATA_OUT is zero, W, STATUS altered            *
; ************************************************************
WRITE_1W movlw     8           ; Loop count
         movwf     COUNT
W_LOOP   bcf       INDF,DAT    ; Low edge signals Start
         Delay_us  1           ; for 1us
         rrf       DATA_OUT,f  ; LSB first shift into Carry
         btfsc     STATUS,C    ; Was it a 1?
          bsf      INDF,DAT    ; IF it was THEN output high
         Delay_us  d'60'       ; Hold for 60us
         bsf       INDF,DAT    ; Release line to go high
         Delay_us  1           ; Relax for 1us
         decfsz    COUNT,f     ; Repeat eight times
         goto      W_LOOP
         return

; ************************************************************
; * FUNCTION: Reads a byte datum from a 1-Wire slave         *
; * RESOURCE: macro Delay_us giving N microsecond delay      *
; * ENTRY   : None                                           *
; * EXIT    : Datum is in DATA_IN, W, STATUS altered         *
; ************************************************************
READ_1W  movlw     8           ; Loop count
         movwf     COUNT
R_LOOP   bcf       INDF,DAT    ; Low edge signals Start
         Delay_us  1           ; for 1us
         bsf       INDF,DAT    ; Release line
         Delay_us  8           ; Wait 8us for Slave to O/P data
         bcf       STATUS,C    ; Clear Carry
         btfsc     INDF,DAT    ; Check input state
          bsf      STATUS,C    ; IF high THEN set Carry
         rrf       DATA_IN,f   ; Shift bit in -> LSB
         Delay_us  d'48'       ; Wait to end of slot
         decfsz    COUNT,f     ; Repeat eight times
         goto      R_LOOP
         return
```

Eight Write slots are used with a 1 μs relax period interval to transmit the byte, each slot's state following the bit rotated into the Carry flag of the datum byte DATA_OUT. After eight shift/output cycles the process terminates.

Reading from a Slave involves the following tasks:

1. The Master starts the process by forcing the data line low for at least 1 μs.
2. The Master then listens to data placed on the line by the Slave which is valid for up to 15 μs after the Start edge.
3. The Slave releases the line after 15 μs which should be pulled high by the end of the 60 μs slot.

4. The Master waits for a minimum of 1 μs before commencing the next slot.

The input subroutine READ_1W follows this task list, sampling the data line sometime before 15 μs into the slot, at which time the Slave's data should have settled to the appropriate voltage level. Each bit is used to set the Carry flag which is then shifted into DATA_IN. After eight sample/shift loops, DATA_IN has the received byte datum.

Unlike the I^2C bus, the 1-Wire architecture is designed for a single Master. However, 1-Wire Slaves have device addresses comprising a 64-bit unique code as part of an internal ROM. The first eight bits are a 1-Wire family code – the DS1820 code is 10*h*. The following 48 bits are a unique serial number and the last eight bits are an error checking byte.

Self-assessment questions

12.1 Rewrite Program 11.3 on page 288 but based on the SPI hardware of Fig. 12.5. Hint: Rather than shifting in whole bytes it may be more efficient to simply shift in and test on a bit-by-bit basis.

12.2 Show how you could connect four MAX518 ADCs (see Fig 12.16) on the one I^2C circuit and how channel 1 on the third ADC could be written to.

12.3 Communications along a 1-Wire link begins with a Reset operation where the Master pulls the line low for 480 – 960 μs after which the line is released. The Slave then responds by dragging the line low after no more than 60 μs delay. This low persists for a further 60 – 240 μs after which the Slave releases this line. Design a subroutine that will do this procedure when called. Assume the resources of Program 12.16 are available to you.

12.4 Parity is a technique whereby the number of digits in a word is always either even or odd. This is accomplished by adding an extra bit which is calculated by the transmission software to be 0 or 1 to meet this overall criterion. For instance, for odd parity of an 8-bit word 01101111 we have **1**01101111. The receiver will check that all nine received bits have an odd count. If one bit (or any odd number) has been corrupted by noise, then a **parity error** is said to have occurred.

Based on the PIC USART, write software to set the asynchronous protocol to 9 bit word and calculate the odd one's parity bit of DATA_OUT which should be placed in TX9D of the TXSTA register prior to the loading of the data into TXREG and transmission.

12.5 Rewrite the subroutine GETCHAR of Program 12.11 as an interrupt service routine called GETCH. Compare the two approaches.

12.6 A certain data logger is to sample temperature once per 15 minutes. The power supply current consumption is reduced by using a PIC16LC74 (Low-voltage) part at a V_{DD} of 3 V and a crystal of 32.780 MHz. Under these conditions the current consumption with the Timer 1 running is a maximum of 70 μA (45 μA typical). A I^2C EEPROM is to be used to store the data as it is read but is only powered on at sample time – by using a spare port line as the EEPROM's power supply. The logger is to be left submerged at the bottom of a lake for six months before being recovered. Can you choose an appropriate 24LCXXX EEPROM and estimate the capacity of the 3 V battery in mA-hours?

12.7 When the data logger alluded to in the last SAQ is brought back to base it is to be connected to a PC in the manner illustrated in Fig. 12.21 and the data uploaded via the serial port. The data terminal running on the PC has set the serial port to 4800 baud with a 8-bit word. The data logger is to transmit an ASCII character for STX (02h) to the PC which if ready is to respond by sending back the code for ACK (00h). After this handshake the logger sends the EEPROM data beginning at address 000h with two ASCII characters representing each stored byte. For example if the byte is A9h then the codes 41h followed by 39h are transmitted; i.e. 'A' '9'. When the logger encounters the EEPROM datum FFh it is to terminate the conversation with the PC by sending the ASCII code for EOT (04h). Using the code of Programs 12.15 and 12.12 as a guide write a suitable program.

12.8 A typical Liquid Crystal Display, for example the Hitachi LM032L, is shown in Fig. 12.25. Show how you could use a PIC16F84 to give the LCD display an I^2C interface.

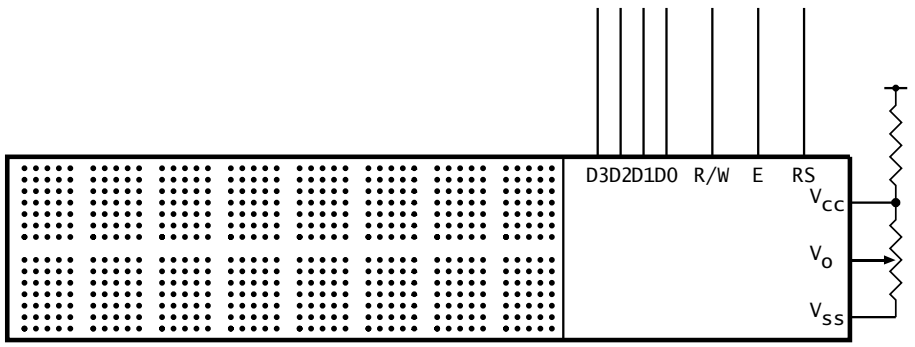


Fig. 12.25 A LCD display.

CHAPTER 13

Time is of the Essence

Of crucial importance in many systems are time-related functions. This may manifest itself in the measurement of duration, event counting or control of an external physical event for known periods. An example of the former would be the time between pulses generated by the teeth on a flywheel to measure engine speed for a tachometer.

Where *time is of the essence* these functions are often best implemented by using hardware counters to time events. In this chapter we will look at the various timer modules which are available to the mid-range PIC family. After completion you should:

- Know how a Watchdog timer improves the robustness of a MCU-based system and how to use the integral PIC device.
- Be able to use the basic 8-bit Timer 0 module as both a counter and timer.
- Understand the function of the 16-bit Timer 1 module and its interaction with the Capture/Compare/PWM (CCP) modules.
- Be able to use the 8-bit Timer 2 module together with the CCP modules to generate a pulse-width modulated output.

Many MCU-based systems are hosted in an electrically hostile environment with noise induced outside both through logic lines and the power supply. Our example of an auto tachometer is typical of this situation, with induction from the high-voltage ignition sparks and alternator sourced ripple in the battery supply. No matter what precautions in shielding and filtering are taken, it is inevitable that on occasion the MCU will jump out of its proper location in Program memory and 'run amok' with potentially serious consequences on the controlled system.[1] In some cases this is little more serious than requiring a manual reset.[2] However, this is not possible in many situations; for example, in a pacemaker implanted in the patient's body!

One solution to this problem is to use a counter/oscillator which resets the processor when it overflows. If the software is arranged to clear this counter on a regular basis so that overflow never occurs then the

[1] The same can happen due to software bugs.

[2] As in a Window's PC.

MCU never resets. If something happens and the MCU jumps out of its normal loop then the counter will overflow and the MCU will be reset to its starting point. This counter circuit is given the name **Watchdog timer**.[3]

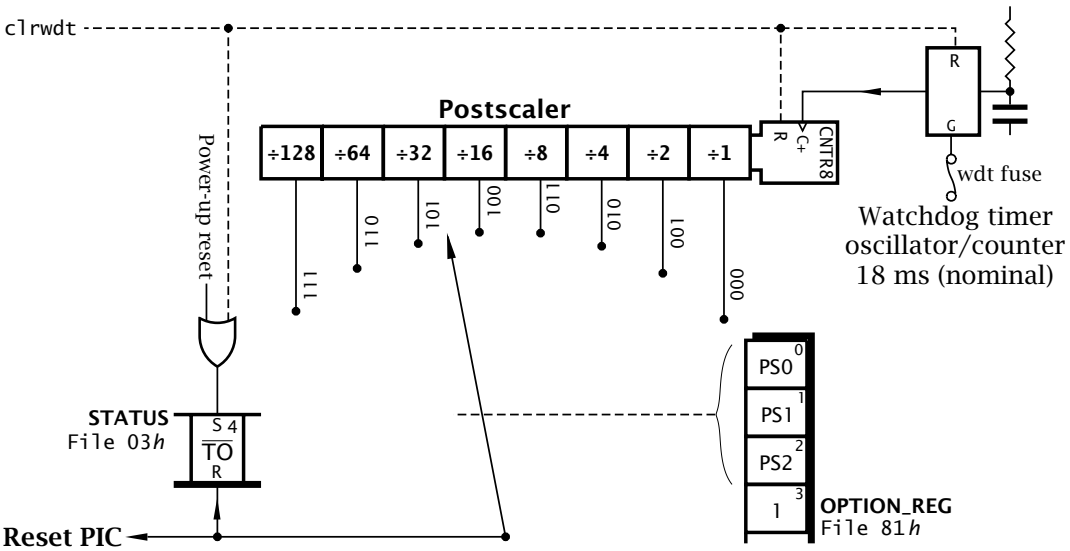


Fig. 13.1 The integral PIC Watchdog timer.

Rather than rely on external Watchdog timers, all PIC MCUs, even the early 12-bit PIC16C5XX family, have an integral module as illustrated in Fig. 13.1. A built-in oscillator *separate* from the processor clock runs continually if the Watchdog timer is enabled, with a nominal period of 18 ms. The integral capacitor/resistor timing network is not tightly specified and as a consequence the period will vary with device process, temperature and supply voltage from a minimum of 7 ms (lowest temperature −40°C, highest V_{DD} 6 V) up to a maximum of 33 ms (highest temperature +85°C, lowest V_{DD} 2 V) – see Fig. 15.8 on page 448.

The Watchdog oscillator is followed by an 8-bit **Watchdog postscaler** counter. This prescaler be set to give Watchdog counter overflow time-out periods going up in powers of two up to nominally $18 \times 128 \approx 2.3$ s – 0.9 s minimum, 4.2 s maximum. The actual value is selected by the programmer using the **PS[2:0]** (**Pre/Post Scaler rate Select**) bits in the **Option register** (**OPTION_REG**) – see Fig. 13.2. The Watchdog postscaler and counter are both cleared with the **`clrwdt`** (**CLeaR WatchDoG Timer**) instruction and issuing this at regular intervals is the mechanism whereby the programmer prevents time-out.

[3] Other approaches typically are based on a retriggerable monostable.

OPTION_REG File 81*h*

7	6	5	4	3	2	1	0
$\overline{\text{RBPU}}$	INTEDG	**T0CS**	**T0SE**	**PSA**	**PS2**	**PS1**	**PS0**
(R/W 1)	(R/W 1)	(R/W 1)	(R/W 1)	(R/W 1)	(R/W 1)	(R/W 1)	(R/W 1)

Fig. 13.2 The Option register.

The Watchdog timer postscaler is a shared resource with Timer 0 (see Fig. 13.3) and cannot be used for both functions. The **PSA** (**Pre/Post Scaler Assignment**) bit is used to assign this resource either to the Watchdog timer or to Timer 0. On Reset the postscaler is assigned to the Watchdog timer and time-out is set to ÷128.

Even with this default assignment the programmer must enable the latter by setting the **WDTE** configuration fuse, as described in Fig. 10.5 on page 261; for example:

```
__config _HS_OSC & _WDT_ON & _PWRTE_OFF & _CP_OFF
```

or the equivalent that is appropriate to your **C** compiler; for instance, in the CCS compiler:

```
#fuses HS, WDT, NOPUT, NOPROTECT
```

If the Watchdog timer overflows it clears the $\overline{\text{TO}}$ (**Time Out**) bit in STATUS[4] – see Fig. 4.5 on page 89. This bit is set by the `clrwdt` or `sleep` instruction and on Power-on reset. `clrwdt` also sets the $\overline{\text{PD}}$ status bit. $\overline{\text{TO}}$ is cleared after Watchdog time-out has occurred; as listed in Table 10.3 on page 266. It is not changed if the PIC is manually reset. $\overline{\text{TO}}$ is a read-only bit and therefore cannot be altered by the programmer other than by using `clrwdt` or by putting the MCU in its Sleep state.

Unless the PIC is sleeping a **Watchdog reset** will force execution to begin at the normal Reset vector 000*h*. In some circumstances the programmer may want to distinguish between the two sources of reset. For example, assume that a system is counting cans of beans moving along a conveyer belt in the manner shown in Fig. 13.4, keeping a tally in a file register called BEAN_COUNT. On Power-on this tally is to be zeroed. If due to a glitch the PIC's Watchdog times out and the PIC resets, then this tally should be left unchanged. To do this we can use code to check the state of $\overline{\text{TO}}$ and take the appropriate action; for instance:

```
      __config _WDT_ON       ; Enable the Watchdog timer
      org      000h          ; The Reset vector
MAIN  btfss    STATUS,NOT_TO ; Was this a Watchdog reset?
       clrf    BEAN_COUNT    ; IF not THEN zero tally
; more initializing code
      clrwdt                 ; Set NOT_TO and reset Watchdog
```

Resetting initializes the Watchdog timer and zeros the postscaler so it's not necessary to issue an extra `clrwdt` as we have done above. However, the initializing process will add to the time it takes to get into the normal main code and the normal loop `clrwdt`, thus an extra `clrwdt` at the end of any initializing code is useful insurance.

In the 12-bit PIC family the Watchdog time-out was the only way short of a Manual reset to awaken a device in the Sleep state. The `sleep` instruction resets the Watchdog and postscaler counters to give a full time-out period when entering the Sleep state. As the Watchdog oscillator is stand-alone it continues to run while the PIC is asleep. After the ordained period a Watchdog reset occurs and the PIC resumes execution with the instruction following `sleep`. If necessary the programmer can determine that the time-out occurred during the Sleep state by examining the $\overline{\text{TO}}$ bit.

The Watchdog awaken mechanism is not so important for the mid- and high-range PIC families. These can be awakened by an external interrupt, and where applicable the Timer 1 and Analog modules, both of which have the option of a self-standing clock oscillator. Where the watchdog is enabled and used in the Sleep mode the designer should be aware that the typical current consumption[4] rises from typically 1.5 μA (24 μA maximum) to 10.5 μA (42 μA maximum). Where long-term battery operation is required (for example, see SAQ 12.6 on page 360) this presents serious problems. Running the processor continually at the lower frequency of 32.768 kHz with the Watchdog disabled takes typically 52 μA (105 μA maximum) in comparison.

The original 12-bit PIC16C5XX family features a basic 8-bit counter/-prescaler which was called a Real-Time Clock/Counter. Although this term was still used in the early 14-bit PIC16CXXX data sheets, the introduction of additional timers lead to the more consistent term **Timer 0** (**TMR0**). However, the term RTCC is still to be found as a relic in older textbooks and software. For example, the CCS **C** compiler sets the Timer 0 clock source to external low-going edge with a prescale value of 4 by calling `setup_counters(rtcc_ext_l_to_h, rtcc_div_4);`.

From Fig. 13.3 we see that Timer 0 comprises a primary 8-bit counter located at File 1 in tandem with an optional 8-bit prescaler counter. This gives eight selectable clock rates into the primary counter as selected by the three PS[2:0] bits in OPTION_REG. This Timer 0 prescaler is actually

[4] The PIC16C74 with V_{DD} = 4 V, −40°C to +85°C.

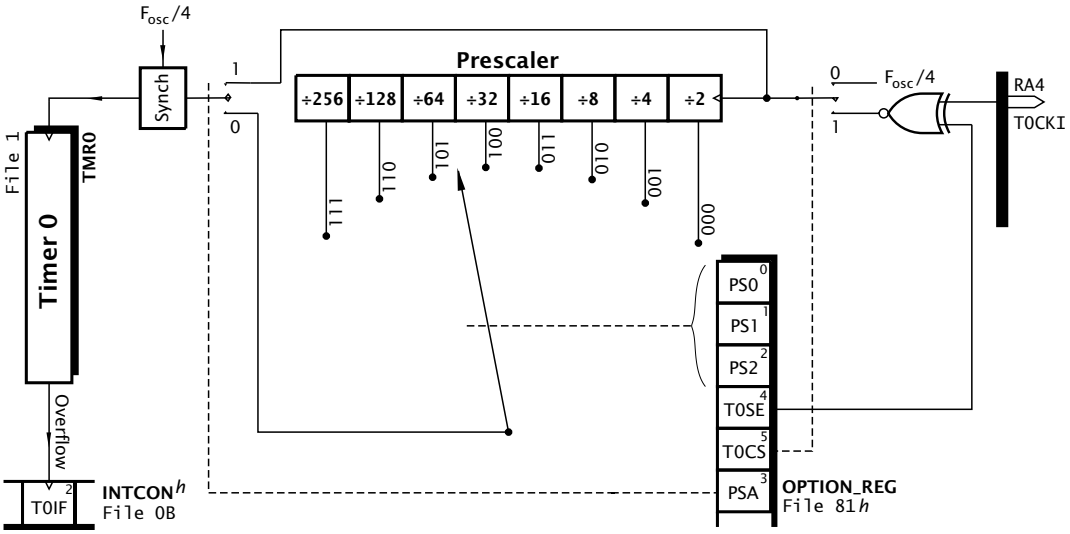


Fig. 13.3 Simplified equivalent circuit for Timer 0.

the same circuit as the Watchdog's postscaler[5] and PSA in OPTION_REG[3] must be set to 0 to assign the prescaler to Timer 0.

The prescaler is assigned to the Watchdog timer by default on Power-on/Manual reset and in this situation the primary counter is either clocked by the internal cycle clock F_{osc} at a rate XTAL/4 or from an outside source via the **Timer 0 Clock Input RA4/T0CKI** pin. The **Timer 0 Clock Select T0SC** bit at OPTION_REG[5] is used to select the internal/external mode. When clocked from outside, the active edge is set using the **Timer 0 Set Edge T0SE** bit at OPTION_REG[4].

In order to synchronize the outside clock to the internal Timer 0 primary counter, a 2-stage shift register is used at the latter's clock input. This causes a $2 \times F_{osc}/4$ delay; 1 μs with an 8 MHz crystal. Where the primary counter is directly connected to the internal clock, this will cause a 2-count delay before anything happens after a datum is written into Timer 0 at File 1. This synchronization is such that Timer 0 may be read from or written to in the normal way without interfering with any possible count action.

When the primary counter overflows (11111111 → 00000000) the **Timer 0 Interrupt Flag T0IF** is set. This event can be sensed by polling INTCON[2] or if the **Timer 0 Interrupt Enable** mask bit **T0IE** in INTCON[5] is set an interrupt will automatically be generated – see Fig. 7.4 on page 178.

An external clock signal going directly into the primary counter should be high for at least $2T_{osc} + 20$ ns and low for at least the same time. Thus for a 8 MHz crystal, T_{high} should be 270 ns and the same for T_{low}; a max-

[5] The PIC18CXXX family use separate scalers for the Watchdog timer and Timer 0.

imum frequency of 1.8 MHz. When the prescaler is used this minimum total period of $4T_{osc}$ + 40 ns can be divided by the prescaler ratio. The input waveform is subject only to a minimum 10 ns pulse width. Thus with a ÷256 setting, a nominally 50 MHz signal at T0CKI will be counted.

When assigned to Timer 0 the prescaler is not readable, so the timer is not strictly a 16-bit counter. Reading Timer 0 does not affect the prescaler but any instruction writing to it (eg. `clrf 1`, `movwf 1`) will both *clear* the prescaler and the clock synchronizer.

As the prescaler is assigned to the Watchdog on Power on/Manual reset it can be subsequently changed over by clearing PSA. However, it is possible that this change-over could cause a Watchdog reset even if it is disabled. Microchip therefore recommend that the change-over be preceded by a `clrwdt` instruction; for example:

```
    clrwdt              ; Clears postscaler and wdt
    bsf    STATUS,RP0   ; Change-over to Bank0
    movlw  b'11110001'  ; External clock on low-going edge
    movwf  OPTION_REG   ; 1:4 Timer0 prescaler
    bcf    STATUS,RP0   ; Back to Bank1
```

The code shows the prescaler divides clock input by four from the T0CKI pin, and incrementing on the ‾_. It is also possible to change the prescaler over from Timer 0 to the Watchdog timer 'on the fly'. In the same manner the `clrwdt` should be executed before altering OPTION_REG to avoid a spurious Watchdog reset.

Timer 0 is mainly used either to count external events or to determine the period between external events. it can also be used to time software toggling port pins for precisely known durations, without tying up the processor in time-wasting delay routines.

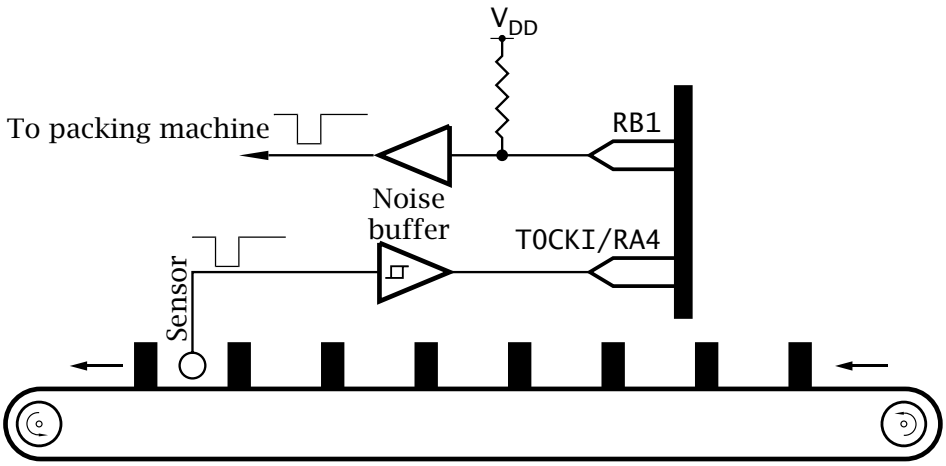


Fig. 13.4 Counting cans of beans on a conveyer belt.

We will illustrate the usage of Timer 0 as an event counter and stop clock with two examples. The first is to tally cans of baked beans travelling along a conveyer belt, as shown in Fig. 13.4. Each 24 cans passing the sensor is to generate a pulse to a packing machine, so that the box can be replaced by a new empty container. This pulse need only be a few microseconds in duration. A double-byte count is also to be kept of the number of boxes packed since the last Power-up/Manual reset. This will be uploaded to the central plant computer at the end of the shift for inventory control.

Our first consideration is the setup and initialization code. The code shown below begins by checking the $\overline{\text{TO}}$ flag at the Reset vector. If zero then the bulk of the initialization code is omitted as reset was due to a Watchdog time-out. If this was not the case then port pin RA4/T0CKI is set up as an input and RB1 set up as an output to activate the packing machine.

```
      include  "p16F84.inc"
      __config _WDT_ON       ; Enable Watchdog

      cblock 20h
       _work:1, _status:1
       COUNT:2
      endc

      org      0             ; Reset vector
      btfss    STATUS,NOT_TO ; Ckeck if a Watchdog reset
       goto    MAIN_LOOP     ; IF yes THEN no initialization
      goto     MAIN          ; ELSE a fresh start

      org      4             ; Interrupt vector
      goto     ISR           ; Foreground program

MAIN  bsf     PORTB,1        ; Idle state of the Packing pulse
      bsf     STATUS,RP0     ; Change to Bank1
      bsf     TRISA,4        ; Make sure that T0CK1 is I/P
      bcf     TRISB,1        ; & RB1/Packing machine an O/P
      movlw   b'00101111'    ; Timer source external -ve edge
      movwf   OPTION_REG     ; prescaler assigned to wdt
      bcf     STATUS,RP0     ; Back to Bank0

      bsf     INTCON,T0IE    ; Enable Timer0 interrupt

      movlw   -d'24'         ; Initialize TMR0 to -24 (E8h)
      movwf   TMR0
      clrf    COUNT+1        ; Clear the 2-byte score count
      clrf    COUNT
      bsf     INTCON,GIE     ; Enable all interrupts

; The background program which amongst other things
MAIN_LOOP
      clrwdt                 ; Regularily resets the wdt
      ...     .....          ; More background code
```

The Option register, also in Bank 1, is set up to assign the prescaler to the Watchdog and extend its time-out period by $\times 128$. The Timer 0 counter is set to be clocked from T0CKI on a negative-going edge. Finally, back in Bank 0 Timer 0 itself is set to E8h (i.e. -24 decimal) so that 24 can pulses will cause it to overflow and cause an interrupt. Both INTCON flags T0IE and GIE are then set to enable the interrupt.

The main background program commences with a `clrwdt` instruction. Provided that the background endless loop is no longer than $7 \times 128 = 0.8961$ s, the minimum Watchdog period, then time-out will not occur.

With the initialization code in situ all that remains is to implement the Interrupt Service Routine (ISR) that will be automaticlly entered after each batch of 24 cans. When this occurs, Timer 0 will set T0IF and the PIC will jump to the Interrupt vector at 004h. In our initialization code we have placed a `goto ISR` at this point and so named the routine in Program 13.1.

Program 13.1 The bean counter Interrupt Service Routine.

```
; ************************************************************
; * The ISR to issue a Packing-machine pulse and             *
; * re-initialize Timer0 to -24.  Also keeps a grand score   *
; * total in COUNT:2 for background analysis                 *
; ************************************************************
; First save context in usual way
ISR     movwf   _work        ; Put away W
        swapf   STATUS,w     ; and the Status register
        movwf   _status

; ************************************************************
; The core code
        btfss   INTCON,T0IF  ; Was it a heartbeat?
         goto   ISR_EXIT     ; IF no THEN false alarm

        bcf     PORTB,1      ; Pulse packing machine
        movlw   -d'24'       ; Re-initialize Timer0
        movwf   TMR0
        incf    COUNT+1,f    ; Add one to score count
        btfsc   STATUS,Z
         incf   COUNT,f
        bcf     INTCON,T0IF  ; Reset interrupt flag
        bsf     PORTB,1      ; End packing machine pulse
; ************************************************************

ISR_EXIT swapf   _status,w   ; Untwist the original Status reg
         movwf   STATUS
         swapf   _work,f     ; Get the original W reg back
         swapf   _work,w     ; leaving STATUS unchanged
         retfie              ; and return from interrupt
```

The ISR itself is sandwiched between the normal context switching wrapper described in Program 7.2 on page 183. The core simply implements the following task list in no particular order:

- Toggle RB1 to signal the packing machine.
- Reset Timer 0 to -24.
- Increment the double-byte score count.
- Reset the Timer 0 Interrupt flag T0IE.

In Program 13.1 T0IE is tested on entry and if not set the ISR is exited. If there are other sources of interrupt then the switch would be to another part of the ISR, as shown in the listing on page 179.

For an alternative approach using hardware interrupts see Program 7.2 on page 183.

Our second example illustrates the use of Timer 0 as a **clock** to measure time between events. The events in question are R-points peaks in the ECG waveform illustrated in Fig. 7.1 on page 172. Here a peak detector interrupts the MCU, which keeps a 2-byte count from a 10 kHz external oscillator. In this manner the period between events can be determined on each event in increments of 100 μs, which we call here **jiffies**. For our example we will modify the specification to eliminate this oscillator and use Timer 0 to keep a 1 ms 2-byte Jiffy tally.

For this task we need to use the main PIC oscillator as the clock source together with the prescaler that so that Timer 0 overflows once per millisecond (10^3 μs). If we choose a 4.096 MHz crystal we have:

$$\text{Time-out} = \frac{4}{4.096} \times 256 \times \text{prescale ratio}$$

which gives a required prescale ratio of 1:4.

With these requirements in mind we have for our initialization background software:

```
        org     0             ; Reset vector
        goto    MAIN          ; Background program

        org     4             ; Interrupt vector
        goto    ISR           ; Foreground program

MAIN    clrwdt                ; Change prescaler to Timer0
        bsf     STATUS,RP0    ; Change to bank1
        movlw   b'00000001'   ; INT on -ve edge, internal clock
        movwf   OPTION_REG    ; prescale div4
        bcf     STATUS,RP0    ; Back to Bank0
        bsf     INTCON,T0IE   ; Enable Timer0 interrupt
        bsf     INTCON,INTE   ; Enable exteral interrupt
        clrf    NEW           ; Zero the New flag
        bsf     INTCON,GIE    ; Enable all interrupts
```

As well as enabling the Timer 0 interrupt, INTE is set to enable hardware interrupts from the INT pin which is going to signal an ECG peak. Neither

Program 13.2 Measuring the ECG waveform period to a resolution of 1 ms.

```
; ************************************************************
; * The ISR to increment the 2-byte COUNT IF TMR0 interrupts  *
; * Copies COUNT:2 to DATA:2 if an INT interrupt and sets NEW *
; * to show background prog that new data is available        *
; ************************************************************
; First save context in usual way
ISR       movwf    _work         ; Put away W
          swapf    STATUS,w      ; and the Status register
          movwf    _status

; ************************************************************

; The core code
          btfss    INTCON,T0IF   ; Was it a heartbeat?
           goto    HEART_BEAT    ; IF yes THEN go to it

          incf     COUNT+1,f     ; Record one more 1ms jiffy
          btfsc    STATUS,Z
           incf    COUNT,f       ; Overflow to upper byte
          bcf      INTCON,T0IF   ; Clear interrupt flag
          goto     ISR_EXIT

HEART_BEAT                      ; Land here if ECG peak
          movf     COUNT+1,w     ; Get new period count LSB
          movwf    DATUM+1       ; Copy into data area
          movf     COUNT,w       ; Get MSB
          movwf    DATUM
          clrf     COUNT+1       ; Zero Jiffy count
          clrf     COUNT
          btfsc    INTCON,INTF   ; Reset interrupt flag
          incf     NEW,f         ; Tell world there is new data

; ************************************************************

ISR_EXIT swapf    _status,w     ; Untwist the original Status reg
         movwf    STATUS
         swapf    _work,f       ; Get the original W reg back
         swapf    _work,w       ; leaving STATUS unchanged
         retfie                 ; and return from interrupt
```

the double-byte Jiffy count nor Timer 0 need be cleared as the first reading of the series will always be erroneous – as the patient's heartbeat is not synchronized to the PIC reset! However, file register NEW which is set to non zero each time an ECG peak is detected is cleared.

The core of the ISR shown in Program 13.2 implements the following task list when an interrupt is received:

1. IF Timer 0 interrupt.
   - Increment 2-byte Jiffy count.

   - Reset Timer 0 interrupt flag.
   - Return from interrupt.
2. ELSE a hardware interrupt form peak picker.
   - Copy jiffy count into general-purpose file registers.
   - Zero Timer 0.
   - Set New flag.
   - Reset hardware interrupt flag.
   - Return from interrupt.

Both bytes in COUNT:COUNT+1 are copied into the two data file registers DATUM:DATUM+1 when a hardware interrupt is received and the Jiffy count/Timer 0 is then zeroed ready for the next event. When the background program polls file register NEW and finds a non-zero datum then it knows that a fresh count is ready for collection. It then, for instance, could send it to a serial EEPROM as in Example 12.3 on page 351 or down a serial link to a PC for subsequent processing and display.

Most mid- and high-range PICs have at least two additional timer/counters and associated circuitry with the following properties.

Timer 1
This 16-bit counter has its own dedicated oscillator and programmable prescaler. Its state can be sampled by an external event and it can control the state of a pin when it reaches a pedefined value.

Timer 2
This 8-bit counter has both programmable pre and postscaler facilities. Its count length can be set by the programmer and it may be used to generate a pulse-width modulated output with no on-going software overhead.

Capture/Compare/PWM
Both timers can be used in conjunction with additional logic called Capture/Compare/Pulse Width Modulation (CCP) to implement the Timer 1 sample instant (Capture), the Timer 1 roll-over value (Compare) and the automatic PWM generation from Timer 2.

Timer 1 comprises a primary 16-bit counter implemented as a pair of file registers at File 0E*h* for the low byte **TMR1L** and File 0F*h* for the high byte **TMR1H**. The **Timer 1 CONtrol register TMR1CON** at File 10*h* configures Timer 1 as shown in Fig. 13.5.

Timer 1 has the option (**T1OSCEN** in T1CON[3] = 1) of using a separate oscillator from the main PIC oscillator. This avoids having to pick the main crystal to suit the timer, as we did in our Timer 0 bean counter example. Some older PIC devices, such as the PIC16C74A, require the RC0/T1CKI pin to be set as input for the oscillator to function. Newer devices, such as the PIC16C74B, do not need this configuration. The Timer 1 oscillator has a maximum frequency of 200 kHz but is typically used with

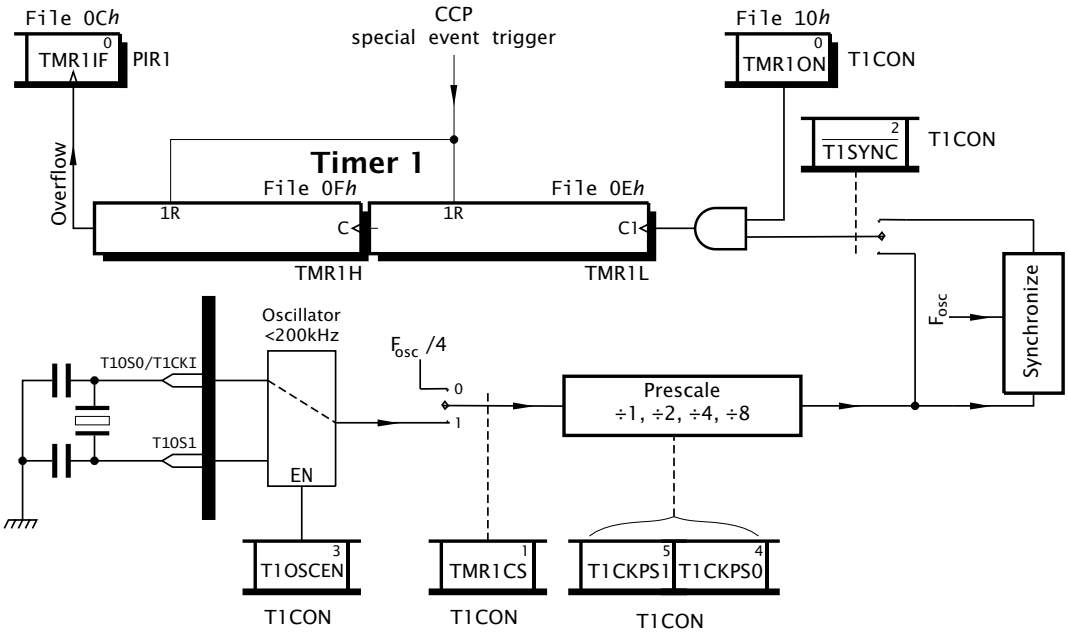


Fig. 13.5 Functional equivalent circuit for Timer 1

a 32.768 kHz watch crystal. Where this is the case, Timer 1 will overflow in 2 seconds with a prescale ratio of 1:1 (**T1CKPS[1:0]** = 00) and a maximum of 16 s for a prescale ratio of 1:8 (T1CKPS[1:0] = 11). When overflow takes place, the **Timer 1 Interrupt Flag** in the Peripheral Interrupt Register 1 PIR1[0] is set. If the corresponding **TMR1IE** mask in the Peripheral Interrupt Enable 1 register PIE1[0] then an interrupt will occur. All interrupt flags and mask bits for Timer 1, Timer 2 and their related CCP modules are located in PIR1, PIR2, PIE1 and PIE2 as shown in Fig 14.10(b) on page 408. To enable *all* these interrupts the **PEIE** (**PEripheral Interrupt Enable**) bit in INTCON[6] must be set as well as the overall GIE global mask bit in INTCON[7]. The latter should be 0 if the only action required is to awaken the PIC from its Sleep state, but PEIE must still be set.

The Timer 1 oscillator adds approximately 20 μa current drain which is a consideration that is especially important if it is intended to use Timer 1 to awaken the processor. Where power consumption is at a premium then a low-power external oscillator should be considered. In this situation with T1OSCEN = 0 the external oscillator should drive the **T1CKI** pin. Limitations on the upper frequency of such an input are similar to that discussed for Timer 0. Alternatively the internal PIC clock can be used if **TMR1CS** is zeroed (the reset condition), but of course this stops when the processor is in its Sleep state.

Output from the programmable prescaler is by default synchronized to the internal clock giving a 2-cycle delay. However, unlike Timer 0 this synchronization shift register can be bypassed with **$\overline{\text{T1SYNC}}$** set to 1.

This needs to be done to allow Timer 1 to operate in the Sleep mode as F_{osc} is disabled in this situation. Apart from this case $\overline{\text{T1SYNC}}$ should be 0 as the lack of synchronization can lead to an unpredictable outcome if data is written into the two Timer 1 primary registers. If the Timer 1 state is to be updated, then the count should be stopped by clearing **TMR1ON** during this process; for instance:

```
    movlw  80h
    bcf    T1CON,TMR1ON    ; Stop the timer
    movwf  TMR1H           ; Set Timer1 to 8000
    clrf   TMR1L
    bsf    T1CON,TMR1ON    ; Restart the timer
```

Altering the state of Timer 1 will always clear the prescale counter.

Timer 1 can be read at any time whatever the state of $\overline{\text{T1SYNCH}}$. However, as only one byte can be read at a time[6] it is possible that the timer may have overflowed from lower to higher byte in between the two reads; for example:

```
; Assume Timer1 is at state 80FFh
    movf    TMR1L,w   ; Read low byte = FFh
    movwf   TEMPL     ; Store away
; «« Timer 1 now increments to state 8100h »»
    movf    TMR1H,w   ; Get high byte = 81h
    movwf   TEMPH     ; Store away
```

erroneously reads the state as 81FF*h* instead of 80FF*h*. This is even more likely to occur if another peripheral device interrupts between reads. A predictable reading can be obtained by either stopping Timer 1 before taking the readings or by reading the high byte first and then checking after the low byte has been read that the high byte has not changed.

The T1CON register is zeroed on Power-on and Manual reset – see Appendix B. This means that Timer 1 defaults to off with an internal clock source and prescale value of 1:1.

For our example assume that we have a low-power temperature logger that is to read the sensor and transmit its value back to base once every 15 minutes. It is proposed that Timer 1 be used to action this process and that the Timer 1 oscillator with a 32.768 kHz watch crystal is to give the timebase.

Timer 1 has a maximum overflow rate of 16 seconds, but if set to 4 seconds we will have a whole number of 15 interrupts per second. If we keep a Jiffy count then a total of $15 \times 15 = 225$ will give 15 minutes. Thus our set up and main skeleton software would be something like that shown in Program 13.3. Here Timer 1 is set up to use the external

[6] In the PIC18CXXX family reading one of the bytes of Timer 1/3 automatically makes a copy of the other byte in a temporary register, effectively giving a single 16-bit time sample.

Program 13.3 Generating a 15 minute data logger timebase.

```
        include "p16C74b.inc"
         __config _WDT_OFF & _CP_OFF

         cblock  20h
          _work:1, _status:1, JIFFY:1
         endc

         org     0
         goto    MAIN
         org     4
         goto    ISR

MAIN     movlw   b'00011011'   ; Timer on, external clock, synched
         movwf   T1CON         ; Oscillator enabled, PS ratio 2:1

         clrf    JIFFY         ; Zero Jiffy count

         bsf     STATUS,RP0    ; To Bank1
         bsf     PIE1,TMR1IE   ; Enable the Timer1 interrupt
         bcf     STATUS,RP0    ; Back to Bank0

DOOZE    sleep                 ; Remember, the 1st instruction
         movlw   d'225'        ; Check, 225 Jiffies = 15 minutes?
         subwf   JIFFY,w
         btfss   STATUS,Z
          goto   DOOZE         ; IF not THEN go back to sleep
         clrf    JIFFY         ; ELSE reset Jiffy count
         call    SAMPLE        ; Sample temperature and transmit
         goto    DOOZE         ; and go back to sleep

; ************************************************************
; First save context in usual way
ISR      movwf    _work        ; Put away W
         swapf    STATUS,w     ; and the Status register
         movwf    _status

; ************************************************************
; The core code
         btfss    PIR1,TMR1IF  ; Was it a Timer1 interrupt?
          goto    ISR_EXIT     ; IF no THEN false alarm

         incf     JIFFY,f      ; Record one more Jiffy
         bcf      PIR1,TMR1IF  ; Reset interrupt flag
; ************************************************************

ISR_EXIT swapf    _status,w    ; Untwist the original Status reg
         movwf    STATUS
         swapf    _work,f      ; Get the original W reg back
         swapf    _work,w      ; leaving STATUS unchanged
         retfie                ; and return from interrupt
```

oscillator with a prescale ratio of 2:1, giving our 4 s jiffy. In addition, both TMR1IE, PEIE and GIE mask bits are set to enable the interrupt on Timer 1 overflow.

The ISR simply adds one onto the Jiffy count. This is tested for 225 in the background program after `sleep` and if equal it is zeroed, the temperature taken and transmitted to base.

Timer 1 can be reset to zero by any **Compare/Compare/PWM CCP** module. Some PIC devices have two CCP modules sharing the same timer, such as the PIC16C74, and in such cases the second module CCP2 is virtually identical to CCP1 and can share the same timer. With this in mind we will look just at CCP1 for convenience, pointing out any differences at the relevant point. All CCP operations require Timer 1 to be configured in its synchronous mode; that is $\overline{\text{SYNCH}} = 0$.

Each CCP module has an associated control register. For CCP1 this is **CCP1CON** at File 1D*h* in which the lower four bits **CCP1M[3:0]** set the module mode. A setting of 0000, the reset value, disables the CCP module, resets the CCP output latch and clears the Capture mode prescaler. Modes 1000 – 1011 listed in Fig. 13.6 give four **Compare modes**. Here an equality comparator detects when the 16-bit Timer 1 datum equals the setting in the 16-bit **CCPR1H:L** (**CCP Register 1**) at File 15:16*h* respectively. When an equality match occurs the **CCP1IF** interrupt flag in PIR1[2] will be set and this can cause an interrupt if the corresponding **CCP1IE** mask bit in PIE1[2] is set.

Besides setting CCP1IF and depending on the setting of the CCP1M[3:0] mode bits, one of four actions are possible on Timer 1 matching CCPR1:

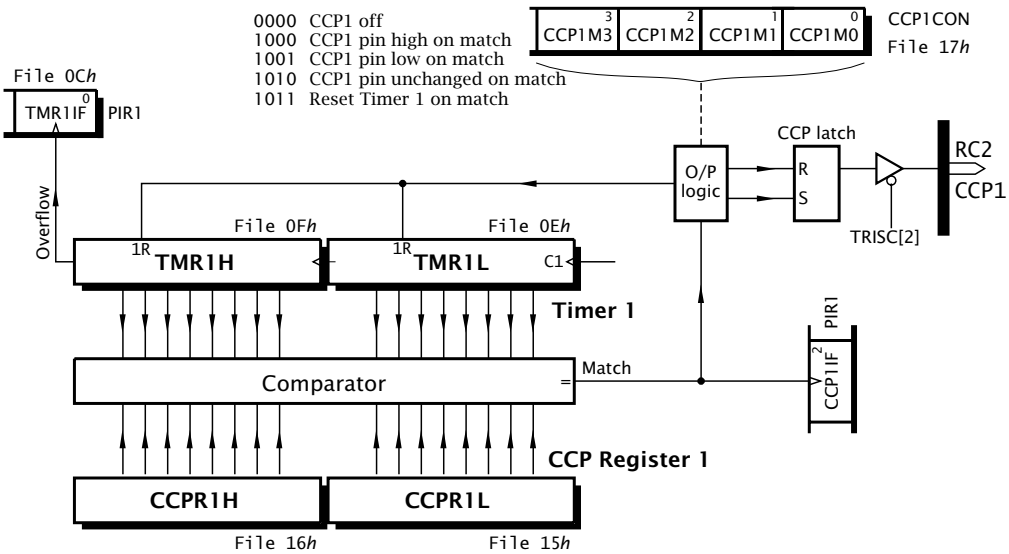


Fig. 13.6 The CCP1 module set to Compare mode.

1000:
Pin RC2/CCP1 is forced high.

1001:
Pin RC2/CCP1 is forced low.

1010:
Pin RC2/CCP1 unchanged, but CCP1IF still set.

1011:
Timer 1 is cleared and with CCP2 only[7] an analog module conversion is initialized by setting GO/$\overline{\text{DONE}}$ – see Fig. 14.8 on page 404.

Where RC2/CPP1 or RC1/CCP2 are to be used as CCP outputs then the appropriate TRISC bit(s) should be cleared to set the pin direction to output. There is no way to directly reset or set the CCP latch other than zeroing the CCPCON register which resets the latch and disables the CCP module. In this case the state of the RC2/CCP1 pin will be that of PORTC[2] until the module is set to an appropriate mode, in which case the pin will reflect the state specified above when the match occurs.

As an example consider that we wish to set up Timer 1 as configured in the last example to generate an interrupt each 10 seconds. To do this we need set the timer to time-out after 16 s (prescale ratio 8:1) and then shorten the cycle. This is implemented by loading the CCPR1 register with the fraction $\frac{10}{16} - 1$, which translates to 9FFFh. Whenever Timer 1 reaches this value it will automatically be reset on the *next* clock input (that would have normally incremented the timer to A000h) and an interrupt will occur if the CCP1IE mask bit (and global PEIE and GIE masks) are set.

Initialization code for this is:

```
movlw  9Fh          ; Set up CCPR1 to 9FFFh
movwf  CCPR1H
movlw  FFh
movwf  CCPR1L
movlw  b'00001011'  ; CCP Compare mode 1011
movwf  CCP1CON
movlw  b'00111011'  ; Timer1 on (1), external clock (1)
movwf  T1CON        ; Synched (0), oscillator (1) 8:1 (111)
bsf    STATUS,RP0   ; Change to Bank 1
bsf    PIE1,CCP1IE  ; Enable CCP1 interrupts
bcf    STATUS,RP0   ; Change back to Bank 0
bsf    INTCON,PEIE  ; Enable Timer/CCP interrupts
bsf    INTCON,GIE   ; Enable all interrupts
```

The PIC will then automatically be interrupted every ten seconds.

As CCP1 is not changed by Compare mode 1011 this pin can be used as a normal Port C input/output independently of the CCP1 module.

Where there are two CCP modules they can work in tandem using different modes, but the timebase will be common – see Example 13.3.

[7] This is the only functional difference between CCP1 and CCP2.

Modes 0100 – 0111 configure the appropriate CCP module to **capture** the state of Timer 1 when an 'event' occurs at the appropriate CCP pin. We can see from Fig. 13.7 that an event can be a falling or rising edge on the RC2/CCP1 pin or every 4th or 16th rising edge according to the CCP1M[3:0] mode bits. This Event prescaler is cleared when the mode bits are set to 0000.

Once a defined event has taken place the 16-bit state of Timer 1 is parallel loaded into the CCP register 1 and CCP1IF set. The processor can then subsequently read this frozen value – that is the time. If Timer 1 is reset after each capture then the sampled datum is the time since the last event. Alternatively, as Timer 1 continues to increment, its captured value can be subtracted from the previous reading to give the difference. As the mode may be altered on the fly, the time between rising and falling edge on CCP1 can be measured by toggling CCP1M[0] between captures. This may cause the CCP1IF flag to be set. To prevent false interrupts, CCP1IE should be cleared before the change-over and CCP1IF after the change-over. Alternatively, the CCP1 module can be used to capture the rising edge and CCP2 the falling edge – see Example 13.3. There is no room for the **CCP2IF** and the associated interrupt mask **CCP2IE** mask bit in PIR1/PIE1. Instead bit 0 of **PIR2**/**PIE2** are pressed into service and in many mid-range processors is the only occupant of these registers.

As our example, consider that we wish to measure the period of our ECG signal with the peak detector connected to pin CCP1. If we assume Timer 1 is clocked by its own 32.768 kHz watch crystal, our set up code is something like this:

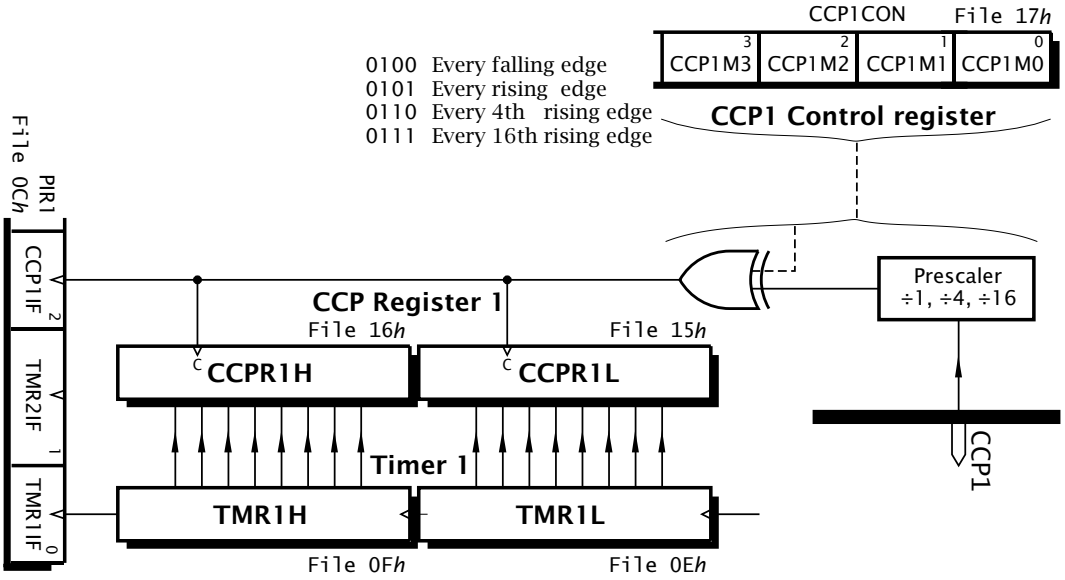


Fig. 13.7 Capturing the time of an event.

```
     movlw   b'00001011' ; Timer on, external clock, synched
     movwf   T1CON       ; Oscillator enabled, PS ratio 1:1

     movlw   b'00000100' ; Capture mode, event = falling edge
     movwf   CCP1CON

     clrf    NEW         ; Zero NEW flag

     bsf     STATUS,RP0  ; To Bank1
     bsf     PIE1,CCP1IE ; Enable the CCP1 interrupt
     bcf     STATUS,RP0  ; Back to Bank0
     bsf     INTCON,PEIE ; Enable Timer/CCP interrupts
     bsf     INTCON,GIE  ; Global interrupts enabled
```

The ISR simply reads the contents of the CCP register and stores it away in two temporary locations, setting the file register NEW to indicate to background program that a new time datum exists. Timer 1 is then reset ready for the next event.

With a crystal of 32.768 kHz the time resolution of the captured datum is 30.5 μs with our 1:1 prescale setting. Timer 1 will overflow in 2 s, which is sufficient to record a heart rate of 30 beats per minute.

Program 13.4 Capturing the instant of time an ECG R-point occurs.

```
; ************************************************************
; First save context in usual way
ISR     movwf    _work        ; Put away W
        swapf    STATUS,w     ; and the Status register
        movwf    _status

; ************************************************************
; The core code
        btfss    PIR1,CCP1IF  ; Was it a CCP1 interrupt?
         goto    ISR_EXIT     ; IF no THEN false alarm

        incf     NEW,f        ; Signal a new capture
        bcf      PIR1,CCP1IF  ; Reset interrupt flag
        movf     CCPR1L,w     ; Get captured low byte
        movwf    TEMP+1       ; Store away
        movf     CCPR1H,w     ; Get captured high byte
        movwf    TEMP         ; Store away
        clrf     TMR1L        ; Zero Timer1
        clrf     TMR1H
; ************************************************************

ISR_EXIT swapf    _status,w    ; Untwist the original Status reg
         movwf    STATUS
         swapf    _work,f      ; Get the original W reg back
         swapf    _work,w      ; leaving STATUS unchanged
         retfie                ; and return from interrupt
```

A more robust software system would also enable the Timer 1 overflow interrupt. if this occurs it indicates that the subsequent captured data will be invalid – although time-outs can be counted and thus extend the validity of the captured time. However, in our system it is more likely to be used to set off an alarm!

Timer 2 is an 8-bit counter which has both pre and postscalers. Unlike the two previous timers, output is not taken from the counter chain but from the **Timer 2 Comparator**. This compares the state of Timer 2 with that in the **Period Register PR2**. On equality an output pulse is generated which resets Timer 2 at the *next* clock pulse. This reset signal also sets the **Timer 2 Interrupt Flag TMR2IF** in PIR1[1]. The number of equality events to set TMR2IF and optionally create an interrupt, is a function of the postscaler. Any integer number between 1 and 16 events can be set up using the **TOUTPS[3:0]** (**Timer 2 OUTput Post Scaler**) in the **Timer 2 CONtrol register** at **T2CON[5:2]**. T2CON at File 12*h* also controls the prescaler, giving four divide ratios for the internal clock, and can also be used to disable the timer. The reset state of T2CON is all zeros, disabling the timer and giving a pre/post scaler ratio of 1:1.

The advantage of this architecture is that time-out can be fine tuned by setting the Period Register to an appropriate value, independently of a CCP module. The period until TMR2IF is set is given as:

$$4/\mathsf{F_{osc}} \times \text{Prescale} \times (\mathsf{PR2} + 1) \times \text{Postscale}$$

For our example, consider the need for an interrupt 100 times per second to scan a 7-segment display (see Program 11.8 on page 302) and that the main crystal is 4 MHz. Choosing a prescale ratio of 4:1 gives a clocking period for Timer 2 of 4 μs. If the Period Register is set to 249 then the Timer 2 comparator output period is $250 \times 4 = 1$ ms. Thus setting

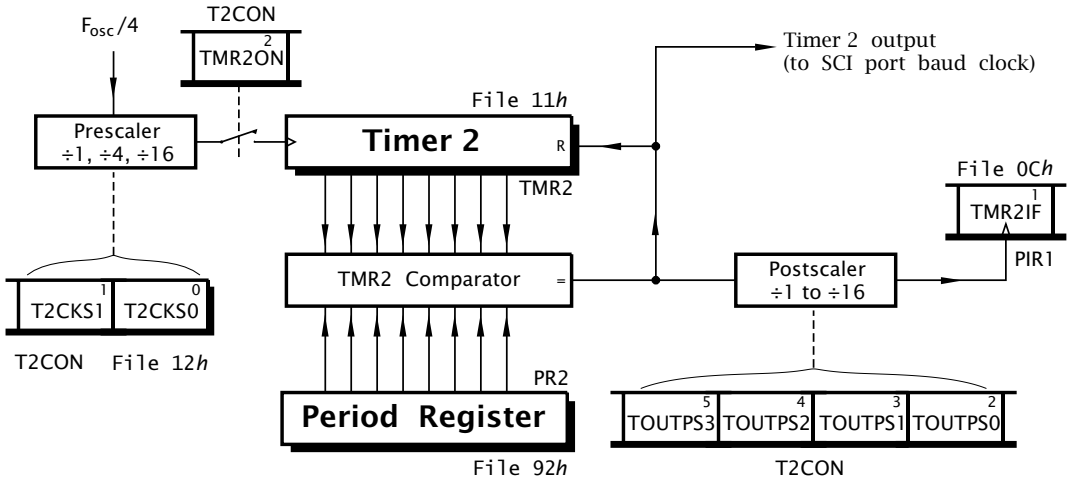


Fig. 13.8 A simplified equivalent circuit for Timer 2.

the postscaler to 1:10 (1001) will give a 10 ms (100 Hz) interrupt rate. By varying the postscaler from 1 to 16 (TOUTPS[3:0] = 0000 to 1111) respectively we can have an interrupt rate from 1 to 16 ms. For fine adjustments a unit change in PR2 alters the rate by 4 × postscaler × μs.

Set up code for this example is:

```
movlw  b'01001101' ; Postscale 1:10 (1001), Timer2 on (1)
movwf  T2CON       ; Prescale 1:4 (01)
bsf    STATUS,RP0  ; Change to Bank1
movlw  d'249'      ; Set up period register to 249
movwf  PR2
bsf    PIE1,TMR2IE ; Enable Timer2 interrupts
bcf    STATUS,RP0  ; Change back to Bank0
bsf    INTCON,PEIE ; Enable all Timer/CCP interrupts
bsf    INTCON,GIE  ; Global enable
```

One of the more common applications of MCU-based systems is the control of power circuits, such as heating, lighting and electric motor speed control. One approach to this problem would be to use a digital to analog converter, such as that discussed in Fig. 12.16 on page 328, driving a power amplifier. Linear control is expensive and inefficient due to the large current:voltage products that must be handled by the power amplifier. A rather more efficient and more cost effective approach rapidly switches the load on and off at a reasonably fast rate. A power switch, such as a thyristor, dissipates relatively little power, as when the switch is off no current flows and when the switch is on the voltage across the switch is low – ideally zero.

An example of such a waveform is shown in Fig. 13.9. The average amplitude is simply $A \times N\%$, where N is the duty cycle percentage of the repeat period. If we vary N from 0 to 100% then the average power will vary in a like fashion – all without the benefit of analog circuitry. This digital to analog conversion technique is known as **pulse width modulation** (**PWM**).

The thermal or mechanical inertia of most high-power loads is such that even with a relatively low repetition rate (typically no lower than 100 Hz) the 'bumps' will be smoothed. Low switching rates are more efficient, as each switching action dissipates energy. If PWM is used for more conventional digital to analog conversion, such as for audio, then a

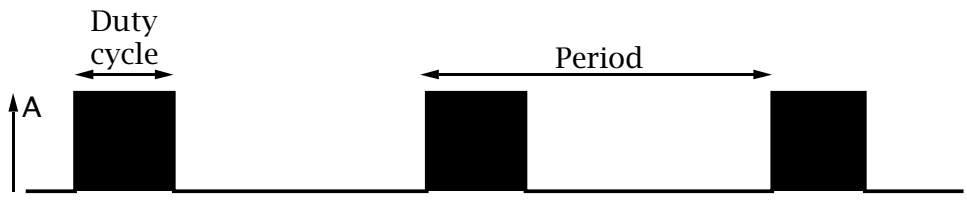


Fig. 13.9 Pulse width modulation.

low-pass filter may be utilized to reduce the high-frequency harmonics. In such cases a sampling rate of at least ten times the maximum analog signal should be used to space out the harmonics (see Fig. 14.3 on page 396) and reduce the necessary filtering burden.

It is relatively easy to generate a PWM waveform solely in software by simply counting and setting a port pin when the count rolls over to zero and resetting the pin when it equals the datum representing the duty cycle. Thus if the duty cycle datum was 9F*h* and the period count was module-256 (00–FF*h*) then the average power would be 62.5% ($\frac{5}{8}$).

CCP modules have a PWM mode in conjunction with Timer 2, as shown in Fig. 13.10. Here Timer 2 runs with a period determined by the main crystal, prescaler and Period Register 2 as previously described. When the Timer 2 comparator causes the count to reset, it also sets the PWM latch. This gives the PWM repeat period.

A second CCP equality comparator matches the 10-bit duty cycle number which is set up by the program in CCPR1L and the two bits **DC1B[1:0]** (**Duty Cycle 1 Bits**) in CCP1CON[5:4]. Taken together this gives a 10-bit duty-cycle datum DC1B[9:0]. This datum is loaded into the CCPR1H and a slave 2-bit latch each time Timer 2 rolls over. CCPR1H is not directly writable to in this mode. Thus we have the following sequence of operations repeated indefinitely:

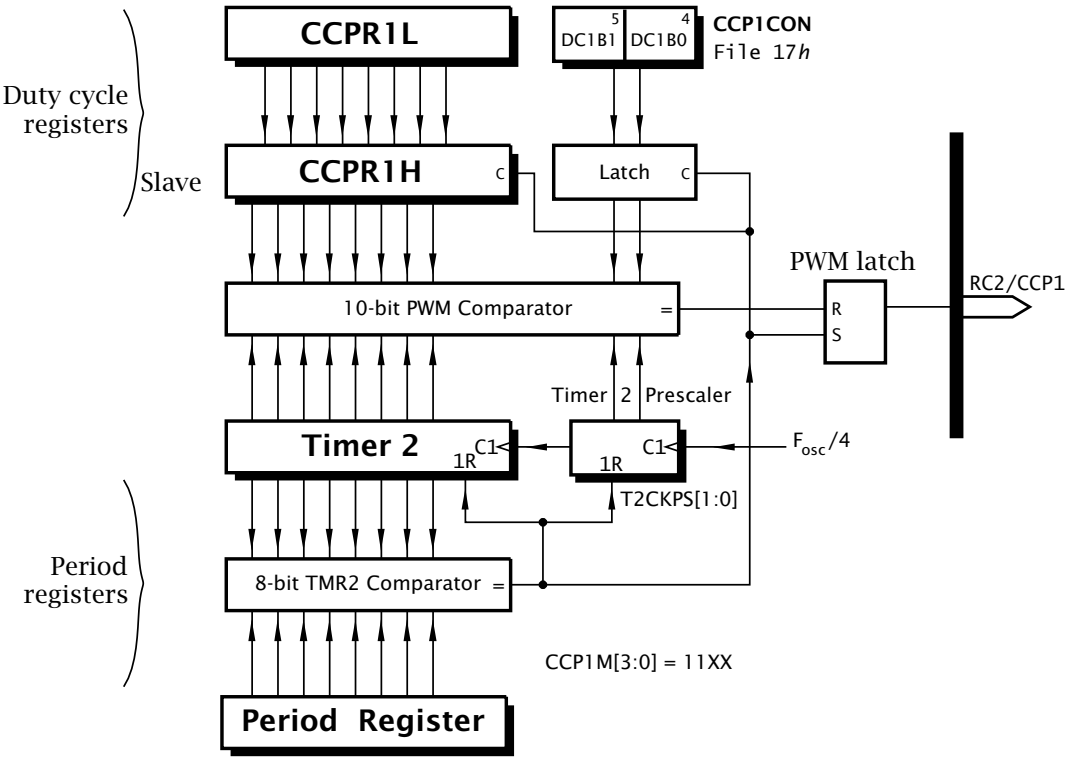


Fig. 13.10 Timer 2 and the PWM CCP mode.

1. Timer 2 increments.
2. When Timer 2:Prescaler equals DC1B[9:0] the PWM latch resets and pin CCP1 goes low.
3. The next clock pulse after Timer 2 reaches the datum in PR2 it is reset. The PWM latch is set, CCP1 goes high and the 10-bit slave duty cycle register/latch us updated.
4. Goto item 1.

From the above description we see that:

- The PWM period is set by the Timer 2/PR2 time-out.
- The duty cycle is set by the 10-bit datum in CCPR1L:CCP1CON[5:4].
- The duty cycle datum can be glitchlessly changed by the software at any time by updating the slave registers and will take effect in the PWM period following this update.
- The RC2/CCP1 pin direction should be set to output.

Where the Timer 2 prescale is set to 1:1 the lower two bits used on the timer side are the quadrature clock phases described in Fig. 4.4 on page 87. This gives a period resolution equal to the crystal period. In all cases the datum in CCPR1L must be smaller than that in PR2, otherwise the PWM latch will never reset!

If PR2 is FFh then the resolution of the system is a full ten bits. Smaller values of Timer 2 period data will reduce this resolution. For example, if PR2 = 3Fh then the resolution is reduced to eight bits – six in PR2 and two in the prescaler.

For our example let us assume a crystal frequency of 20 MHz, a prescale ratio of 1:16 and a PR2 value of FFh. In this case the PWM frequency is $\frac{20}{4} \times \frac{1}{16} \times 256 = 1.22$ kHz. A prescale ratio of 1:1 would increase the period frequency to 19.53 kHz. Reducing the value in PR2 would also increase the repetition frequency. The programmer need only place the duty cycle datum in CCPR1L and CCP1CON[5:4] (the latter can be left at its reset value of zero if the datum is to be treated as 8-bit) and a PWM signal will automatically be generated with no software overhead. Set up code for this instance would be:

```
    bsf   STATUS,RP0    ; Change to Bank0
    movlw FFh           ; Set up Timer2 Period register
    movwf PR2           ; to FFh
    bcf   TRISC,2       ; Make RC2/CCP1 an output
    bcf   STATUS,RP0    ; Change back to Bank1
    movlw b'00001100'   ; CCP1 module PCM mode (1100)
    movwf CCP1CON
    movlw b'00000110'   ; Timer2 prescale 16:1 (10),
    movwf T2CON         ; Timer2 on (1)
```

The Timer 2 postscaler does not affect the PWM generation but still sets the TMR2IF in the normal way. The CCP1IF is not altered in this mode.

If a second CCP module is used, an extra PWM output at pin RC1/CCP2 is available with a separate duty cycle but an identical period, as Timer 2 is a shared resource.

Examples

Example 13.1

Show how you could use Timer 0 to generate a PWM version of a digital byte in file register DATUM using pin RA0 as the output. Assuming an 8 MHz crystal, calculate the PWM duration.

Solution

Timer 0 will give a time-out related to a number loaded into the timer at the beginning of the period. If we load in the the 2's complement of the byte (the negative value) then the duration will be proportional to this value – the larger it is the longer the timer has to count before overflowing. Conversely loading in the value of DATUM will give a time-out duration inversely proportional to the value. By alternately loading the 2's complement of DATUM and making the pin high followed by DATUM itself making the pin low will give us a total period approximately the same as a total Timer 0 time-out as if counting through all 256 states.

The coding of Program 13.5 sets up Timer 0 to count 2 MHz internal clock cycles with no prescale. Thus the total PWM rate is $\frac{2}{256}$ or 7.8125 KHz. When Timer 0 overflows it generates an interrupt. The ISR checks the state of PORTAA[0] and if 0 changes its state and then calculates the 2's complement of the data byte (invert plus one). However, there is a 2-cycle delay in Timer 0 responding when its state is changed due to the clock synchronizer circuit and so another two is added to compensate for this extra delay. If the port pin was already 1 then it is zeroed and the datum itself plus the compensatory two is written into Timer 0.

Adding the compensation will cause problems at either extremes of the mark:space ratio. Why is this so and what action could you take to ameliorate it?

Example 13.2

A certain tachometer is to register engine speed in the range 0–12,000 rpm (revolutions per second). The engine generates one pulse per revolution and it is intended that a PIC16C74 be used to count the number of pulses each second and calculate the equivalent rpm. Using two of the three available timers can you design a suitable hardware-software configuration?

Solution

Program 13.5 Pulse-Width Modulation using Timer 0.

```
MAIN    bsf     STATUS,RP0   ; Change into Bank 1
        clrf    OPTION_REG   ; Internal clock, 1;1 prescale
        bcf     TRISA,0      ; Make RA0 the PWM output
        bcf     STATUS,RP0   ; Change back to Bank 0
        bsf     INTCON,T0IE  ; Enable Timer 0 interrupt
        bsf     INTCON,GIE   ; Enable all interrupts

; <<<< More background code >>>>

; ************************************************************
; * The ISR to generate a PWM waveform at RA0                *
; * Digital byte is in DATUM. PORTA[0] holds current PWM state*
; ************************************************************
; First save context in usual way
ISR     movwf   _work        ; Put away W
        swapf   STATUS,w     ; and the Status register
        movwf   _status

; ************************************************************
; The core code
        btfss   INTCON,T0IF  ; Has Timer0 overflowed?
         goto   ISR_EXIT     ; IF no THEN false alarm

        bcf     INTCON,T0IF  ; Reset interrupt flag
        movf    DATUM,w      ; Get datum
        btfsc   PORTA,0      ; Is current output low?
         goto   MAKE_LO      ; IF not THEN bring it low
MAKE_HI bsf     PORTA,0
        xorlw   b'11111111'  ; ELSE compute 2's complement
        addlw   1            ; Invert +1
        goto    SET_UP       ; and set Timer0 up

MAKE_LO bcf     PORTA,0      ; Bring pin low

SET_UP  addlw   2            ; Compensation for synch delay
        movwf   TMR0         ; Initialize Timer

; ************************************************************

ISR_EXIT swapf   _status,w   ; Untwist the original Status reg
         movwf   STATUS
         swapf   _work,f     ; Get the original W reg back
         swapf   _work,w     ; leaving STATUS unchanged
         retfie              ; and return from interrupt
```

A speed of 12,000 rpm translates to a maximum pulse count of 200 rps (revolutions per second). Thus we propose to use Timer 0 as the pulse counter driven from pin T0CKI with no prescaler.

Timer 1 in conjunction with CCP1 set to a Compare mode will give a 1-second time-out if it uses its own oscillator and a 32.768 kHz watch crystal together with a count spanning 0000–7FFFh. However, to simplify the mathematical relationship rpm = rps $\times$ 60 it is proposed to shorten the timebase by the factor $\frac{60}{64}$ to implement the equivalent relationship $\frac{\text{rps}\times 60}{64} \times 64$. The final $\times$64 can easily be implemented by either shifting left six times (>>6) or more efficiently placing the rps count as the high byte of the double-byte rpm datum and shifting right twice; i.e.:

$$\text{rpm} = (\text{rps} \times 256) >> 2$$

This is considerably more efficient that using a 1 second timebase and multiplying by 60.

One possible solution is shown in Program 13.6. Here the initialization code implements the following task list:

- Set Timer 0 to count ‾_ events at T0CKI.
- Set CCP1 to Compare mode 1011 to reset Timer 1 on equality.
- Enable an interrupt from this event.
- Set up the CCPR1H:L to set the timebase to $\frac{60}{64}$ s.

The ISR itself simply copies the rps reading from Timer 0, zeroing it and then converts the reading to rpm as described above. After the two shift right operation >>2, the top two bits of RPM are cleared to remove erroneous carry-ins. The resulting 14-bit datum in RPM:RPM+1 is the required outcome which can then perhaps be used by the background program to activate the display or maybe transmit the data to a computer over a serial link.

Example 13.3

A PIC16C74 is to be used to measure the duration of an event. This duration is the time a signal is high, as shown in Fig. 13.11. You an assume that the main crystal is 8 MHz and the duration of the event is guaranteed to be no more than 100 ms.

Solution

One way of tacking this problem is to feed the signal shown in the diagram into both pins RC1 and RC2 in parallel. Using one CCP module to capture the rising edge and the other to capture the falling edge gives the duration as the difference in the two captured values. In Program 13.7 Timer 1 is zeroed on a rising edge and thus the second captured Timer 1 state is our duration. If we use a prescale ratio of 1:4 and the internal clock

Program 13.6 Tachometer software.

```
MAIN movlw   77h              ; Setting 77FFh to give
     movwf   CCPR1H           ; a time base of 60/64 seconds
     movlw   0FFh
     movwf   CCPR1L

     bsf     STATUS,RP0       ; To Bank1
     movlw   b'00111000'      ; Timer0 external -ve edge
     movwf   OPTION_REG       ; No prescale
     bsf     PIE1,CCP1IF      ; Enable interrupts from CCP1
     bcf     STATUS,RP0       ; Back to bank0

     movlw   b'00001011'      ; CCP Compare mode 1011
     movwf   CCP1CON          ; resets Timer1

     movlw   b'00001011'      ; Timer1 PS1:1, oscillator synched
     movwf   T1CON            ; and enabled

     bsf     INTCON,PEIE      ; Enable Timer/CCP interrupts
     bsf     INTCON,GIE       ; Global enable mask bit on

; <<<< More background code >>>>

; ************************************************************
; First save context in usual way
ISR     movwf    _work        ; Put away W
        swapf    STATUS,w     ; and the Status register
        movwf    _status

; ************************************************************
; The core code
        btfss    PIR1,CCP1IF  ; Did CCP register Timer1 reset?
         goto    ISR_EXIT     ; IF no THEN false alarm

        movf     TMR0,w       ; Get totalized pulse count
        clrf     TMR0         ; Zero pulse count
        movwf    RPM          ; Save totalized count away

; Now multiply by 64
        clrf     RPM+1        ; Clear lower byte
        rrf      RPM,f        ; RPM as MSB; i.e. X256
        rrf      RPM+1,f      ; >>2 to convert rps to rpm
        rrf      RPM,f
        rrf      RPM+1,f
        bcf      RPM,7        ; Zero top two bits
        bcf      RPM,6

        bcf      PIR1,CCP1IF  ; Reset interrupt flag
; ************************************************************

ISR_EXIT swapf    _status,w   ; Untwist the original Status reg
         movwf    STATUS
         swapf    _work,f     ; Get the original W reg back
         swapf    _work,w     ; leaving STATUS unchanged
         retfie               ; and return from interrupt
```

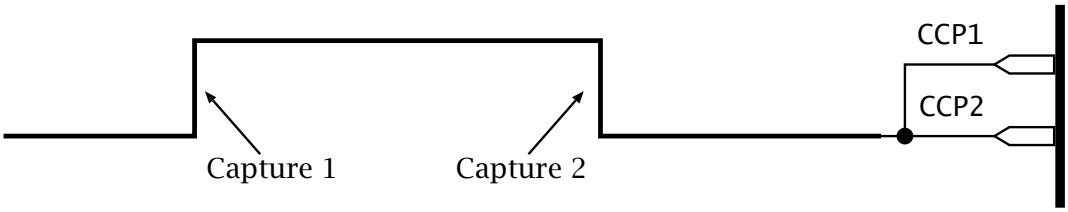


Fig. 13.11 An event manifesting itself as a pulse duration.

then we have as our counting rate 500 kHz; i.e. the system resolution is 2 μs. The overall maximum duration that can be measured in this way is $2^{16} \times 2 = 131,077\mu$s which is large enough not to overflow.

The ISR in Program 13.7 simply tests each CCP interrupt flag in turn and goes to the appropriate routine. If CCP1 has signalled a _/‾ event then the timer is zeroed to restart the count. Timer 1 has been configured to increment at a 500 kHz rate and when the next ‾_ occurs the CCP2 module captures the state of this timebase and places it in the 16-bit `CCPR2` register. The ISR then copies it into the two file registers `TIME:TIME+1` and this is the period in 2 μs ticks.

Actually resetting Timer 1 on the first event introduces some inaccuracy into the process as the clearing event takes some time. In our application this is of little consequence but it may cause some problems in shorter high-resolution situations. In this case Timer 1 can be left to run continually and the two captured 16-bit data subtracted to give the required difference at relative leisure.

Self-assessment questions

13.1 Using Timer 1 and `CCP1`, design a system to generate a continuous square wave with a total period of 20 ms from `RC2/CCP1`. You may assume that the main crystal is 8 MHz. Hint: Remember that the state of the CCP pin will only change when a match occurs so the Compare mode will have to be changed on the fly each 10 ms.

13.2 The echo sounding hardware shown in Fig. 7.6 on page 195 uses an external 1.72 kHz oscillator to interrupt the PIC once per 5.813 ms, that is once every time sound travels 1 cm through air. Assuming that a 20 MHz PIC is used, show how Timer 2 could be used to generate this interrupt rate to an accuracy better than 0.1%.

13.3 The PIC family has only one hardware input, namely `INT/RB0`. Suggest some way to use Timer 0 to simulate another hardware interrupt with pin `T0CKI/RA4`.

Program 13.7 Measuring the duration of a pulse.

```
MAIN movlw   b'00000101'      ; CCP1 module captures rising edge
     movwf   CCP1CON
     movlw   b'00000100'      ; CCP2 module captures falling edge
     movwf   CCP2CON

     bsf     STATUS,RP0       ; To Bank1
     bsf     PIE1,CCP1IE      ; Enable interrupts from CCP1
     bsf     PIE2,CCP2IE      ; Enable interrupts from CCP2
     bcf     STATUS,RP0       ; Back to bank0

     movlw   b'00100001'      ; Timer1 enabled (1), int osc (0)
     movwf   T1CON            ; Synched (0), prescale 2:1 (10)

     clrf    NEW              ; Clear the New flag

     bsf     INTCON,PEIE      ; Enable Timer/CCP interrupts
     bsf     INTCON,GIE       ; Global enable mask bit on

; <<<< More background code >>>>

; ***********************************************************
; First save context in usual way
ISR     movwf    _work        ; Put away W
        swapf    STATUS,w     ; and the Status register
        movwf    _status

; ***********************************************************
; The core code
        btfsc    PIR1,CCP1IF  ; A CCP1 rising edge capture?
         goto    CAPTURE1     ; IF yes THEN go to it!
        btfss    PIR2,CCP2IF  ; A CCP2 falling edge capture?
         goto    ISR_EXIT     ; IF not THEN false alarm!
CAPTURE2
        movf     CCPR2L,w     ; Get low byte of captured time
        movwf    TIME+1       ; and put away
        movf     CCPR2H,w     ; Get high byte of captured time
        movwf    TIME         ; and put away
        bcf      PIR2,CCP2IF  ; Clear flag
        incf     NEW,f        ; Tell the world: A new time datum
        goto     ISR_EXIT

CAPTURE1
        clrf     TMR1L        ; Zero time count
        clrf     TMR1H
        bcf      PIR1,CCP1IF  ; Reset interrupt flag
; ***********************************************************

ISR_EXIT swapf    _status,w   ; Untwist the original Status reg
         movwf    STATUS
         swapf    _work,f     ; Get the original W reg back
         swapf    _work,w     ; leaving STATUS unchanged
         retfie               ; and return from interrupt
```

13.4 As part of a software implementation of an asynchronous serial channel running at 300 baud a delay of $3.\dot{3}$ ms is to be generated. Assuming that a 8 MHz PIC16F84 is the host processor, show how you could use a timer to generate an interrupt each baud period. Extend your routine to enable baud rates up to 19,200 in doubling geometric progression.

13.5 Show how you would use Timer 1 with its separate integral oscillator with a 32.768 kHz watch crystal, to keep the central heating real time clock array `HOURS:MINUTES:SECONDS` of Example 7.4 on page 191 up to date.

13.6 The CCS **C** compiler has integral functions dealing with the timers and CCP modules. For example, Timer 1 can be written to using `set_timer1(datum);` and read from using `get_timer1();`. The function `setup_timer_1(mode);` is used to initialize the timer. Similarily `setup_ccp1(mode);` initializes the `CCP1CON` register. Mode values for Timer 1 and the CCP Compare configuration are:

```
T1_DISABLED          T1_INTERNAL   T1_EXTERNAL
T1_EXTERNAL_SYNCH    T1_CLK_OUT    T1_DIV_BY_1
T1_DIV_BY_2          T1_DIV_BY_4   T1_DIV_BY_8
CCP_COMPARE_RESET_TIMER
```

Where separate modes can be separated by the operator `|`.

Show how you would code your solution to SAQ 13.5 in **C**. In CCS **C** a function can be turned into a `CCP1` interrupt service routine by preceeding it by the directive `#INT_CPP1` – see Program 14.9 on page 427 for details. You an also assume that the reserved variable `CCP_1` represents the 16-bit `CCPR1H:L` register.

CHAPTER 14

Take the Rough with the Smooth

Given that digital microcontrollers are in the business of monitoring and controlling the real environment — which is commonly analog in nature — we need to consider the interconversion between the analog and the digital world. Analog input signals need conversion to a digital equivalent, that is **analog to digital conversion** (**ADC**). Thereafter the digital patterns can be processed in the normal way. Conversely, if the outcome is to be in the form of an analog signal, then a **digital to analog conversion** (**DAC**) stage will be necessary.

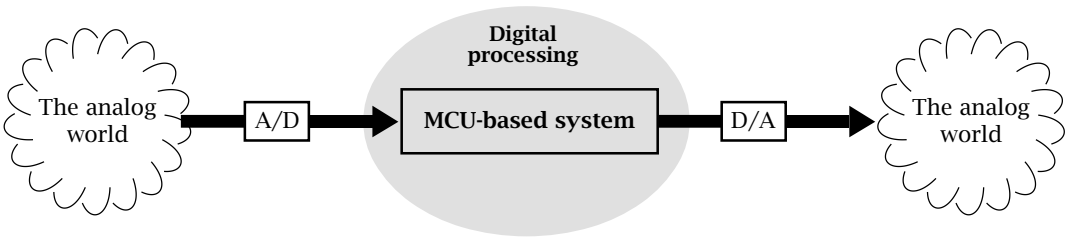


Fig. 14.1 Analog world – digital processing.

Of these two processes, illustrated in Fig. 14.1, A/D conversion is by far the more complex. Some PIC devices, notably the PIC16C7XX and 12C67X lines, feature integral multi-channel A/D facilities. However, analog outputs require external circuitry to implement the D/A process.

In this chapter we will look at the properties of analog and digital signals and the conversion between them as relevant to the PIC MCU. After completion you will:

- Understand the quantization relationship between analog and digital signals.
- Appreciate the need to sample an analog signal at least twice the highest frequency component.
- Appreciate how the successive approximation technique can convert an analog voltage to a binary equivalent.
- Be able to select the correct ADC clocking source and frequency.

- Be able to select the analog channel for conversion.
- Be able to configure I/O pins as either analog or digital.
- Be able to write assembly-level programs to acquire analog data using polling, interrupt-driven and Sleep techniques.
- Be able to code high-level **C** programs to interface to the analog module.
- Know how to interface in parallel to a proprietary DAC.

The information content of an **analog signal** lies in the continuously changeable worth of some constituent parameters, such as amplitude, frequency or phase. Although this definition implies that an analog variable is a continuum between $\pm\infty$, in practice its range is restrained to an upper and lower limit. Thus a mercury thermometer may have a continuous range between, say, -10°C and $+180^{\circ}$C. Below this the mercury disappears into the bulb. Above and the top of the tube is blown off!

Theoretically the quantum mature of matter sets a lower limit to the smooth continuous nature of things. However, in practice noise levels and the limited accuracy of the device generating the signal sets an upper limit to the resolution that processing needs to take account of.

Digital signals represent their information content in the form of arrangements of discrete characters. Depending on the number and type of symbols making up the patterns, only a finite totality of value portrayals are possible. Thus in a binary system, an n-digit pattern can at the most represent 2^n levels. Although this grainy view of the world seems inferior to the infinity of levels that can be represented by an analog equivalent, the quantizing grid can be tailored to be fit for the accuracy of the task to be undertaken. For example, a telephone speech circuit will tolerate a resolution of around 1%. This can use an 8-bit depiction, which gives up to 256 discrete values — $\approx 0.5\%$. A music compact-disk uses a 16-bit scheme, giving a one part in 65,636 grid — an $\approx 0.0015\%$ resolution.

From this discussion it can be seen that any process involving interconversion between the analog and digital domains will involve transition through the **quantization** state. Therefore we need to look at how this affects the information content of the associated signals.

As an example, consider the situation shown in Fig. 14.2, where an input range is represented as a 3-bit code. In essence the process of quantizing a signal is the comparision of the analog value with a fixed number of levels – eight in this case. The nearest level is then taken as expressing the original in its digital equivalent. Thus in Fig. 14.2 an input voltage of 0.4285 of full scale is 0.0536 above quantum level 3. Its quantized value will then be taken as level 3 and coded as $011b$ in our 3-bit system.

The residual error of -0.0536 will remain as quantizing noise, and can never be eradicated (see Fig. 14.3(d)). The distribution of quantization error is given at the bottom of Fig. 14.2, and is affected only by the number

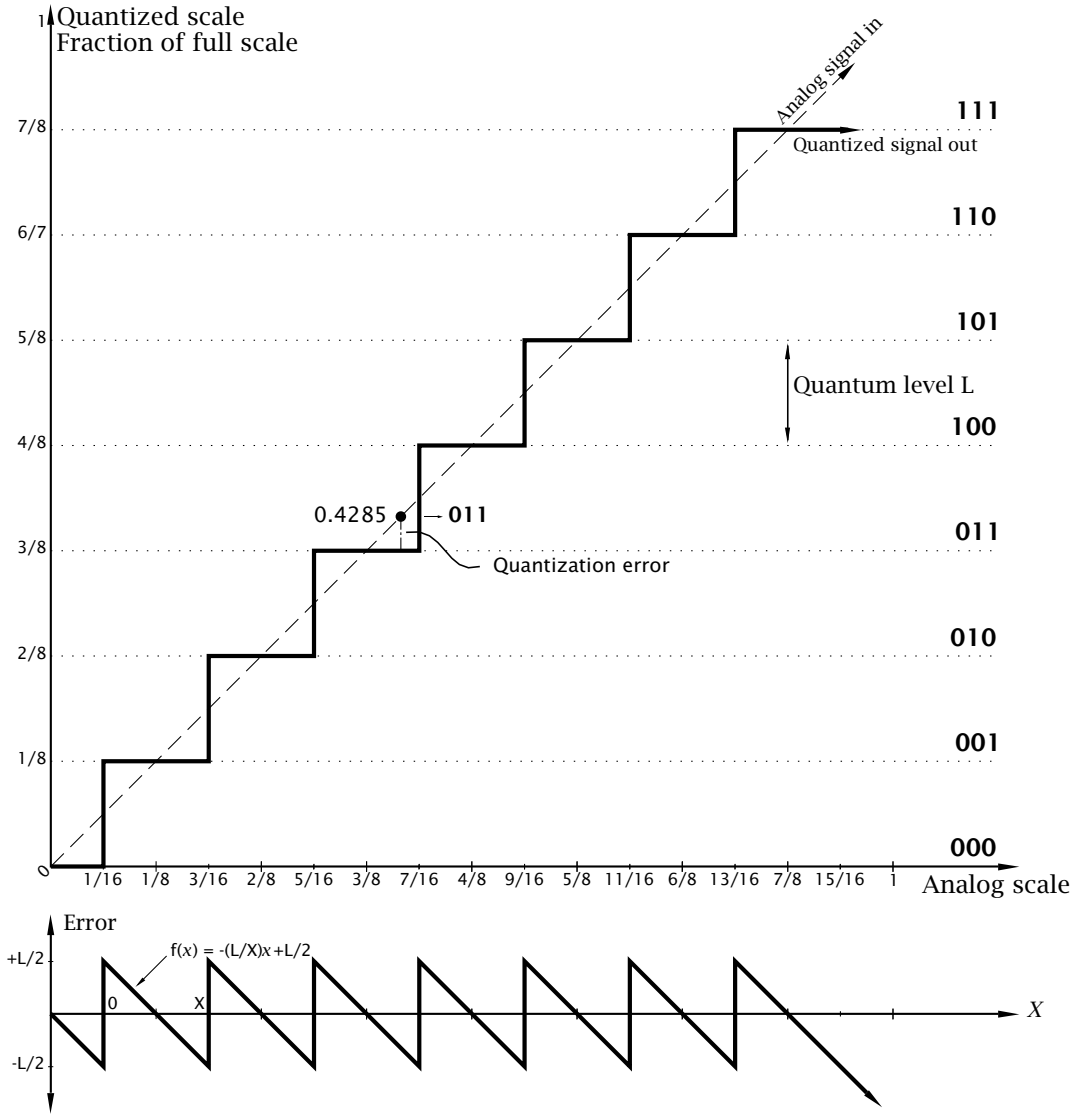


Fig. 14.2 The quantizing process.

of levels. This can simply be calculated by evaluating the average of the error function squared. The square root of this is then the root mean square (rms) of the noise.

$$\mathcal{F}(x) = -\frac{L}{X}x + \frac{L}{2}$$

The mean square is:

$$\frac{1}{X}\int_0^X \mathcal{F}(x)^2\,dx = \frac{1}{X}\int_0^X \left[\frac{L^2}{X^2}x^2 - \frac{L^2}{X}x + \frac{L^2}{4}\right] dx$$

$$= \frac{1}{X}\left|\frac{L^2}{3X^2}x^3 - \frac{L^2}{2X}x^2 + \frac{L^2}{4}x\right|_0^X = \frac{L^2}{12}$$

Thus the rms noise value of $\frac{L}{\sqrt{12}} = \frac{L}{2\sqrt{3}}$, where L is the quantum level.

A fundamental measure of a system's merit is the signal to noise ratio. Taking the signal to be a sinusoidal wave of peak to peak amplitude 2^nL (see Fig. 14.3), we have an rms signal of $\frac{\left(\frac{2^nL}{2}\right)}{\sqrt{2}}$, that is $\frac{\text{peak}}{\sqrt{2}}$. Thus for a binary system with n binary bits, we have a signal to noise ratio of:

$$\frac{\left(\frac{2^nL}{2\sqrt{2}}\right)}{\left(\frac{L}{\sqrt{12}}\right)} = \frac{2^n\sqrt{12}}{2\sqrt{2}} = 1.22 \times 2^n$$

In decibels we have:

$$\text{S/N} = 20\log 1.22 \times 2^n = 6.02n + 1.77\,\text{dB}$$

The dynamic range of a quantized system is given by the ratio of its full scale (2^nL) to its resolution, L. This is just 2^n, or in dB, $20\log 2^n = 20n\log 2 = 6.02n$. The percentage resolution given in Table 14.1 is of course just another way of expressing the same thing.

Table 14.1: Quantization parameters.

Binary bits n	**Quantum levels** (2^n)	**% resolution**	**Resolution Dynamic range**	**S/N ratio (dB)**
4	16	16.25	24.1 dB	26.9 dB
8	256	0.391	48.2 dB	49.9 dB
10	1024	0.097	60.2 dB	61.9 dB
12	4096	0.024	72.2 dB	74.0 dB
16	65,536	0.0015	96.3 dB	98.1 dB
20	1,048,576	0.00009	120.4 dB	122.2 dB

The exponential nature of these quality parameters with respect to the number of binary-word bits is clearly seen in Table 14.1. However, the implementation complexity and thus price also follows this relationship. For example, a 20-bit conversion of 1 V full scale would have to deal with quantum levels less than 1 μV apart. Pulse-code modulated telephonic links use eight bits, but the quantum levels are unequally spaced, being closer at the lower amplitude levels. This reduces quantization hiss where conversations are held in hushed tones! Linear 8-bit conversions are suitable for most general purposes, having a resolution of better than $\pm\frac{1}{4}$%. Actually video looks quite acceptable at a 4-bit resolution, and music can just be heard using a single bit – i.e. positive or negative!!

S/N ratios presented in Table 14.1 are theoretical upper limits, as errors in the electronic circuitry converting between representations and aliasing (discussed below) will add distortion to the transformation.

The analog world treats time as a continuum, whereas digital systems sample signals at discrete intervals. Shannon's sampling theorem[1] states that provided this interval does not exceed half that of the highest signal frequency, then no information is lost. The reason for this theoretical twice highest frequency sampling limit, called the Nyquist rate, can be seen by examining the spectrum of a train of amplitude modulated pulses. Ideal impulses (pulses with zero width and unit area) are characterized in the frequency domain as a series of equal-amplitude harmonics at the repetition rate, extending to infinity. Real pulses have a similar spectrum but the harmonic amplitudes fall with increasing frequency.

If we modulate this pulse train by a baseband signal $A \sin \omega_f t$, then in the frequency domain this is equivalent to multiplying the harmonic spectrum (the pulse) by $A \sin \omega_f t$, giving sum and different components thus:

$$A \sin \omega_f t \times B \sin \omega_h t = \tfrac{AB}{2}(\sin(\omega_h + \omega_f)t + \sin(\omega_h - \omega_f)t)$$

More complex baseband signals can be considered to be a bandlimited (f_m) collection of individual sinusoids, and on the basis of this analysis each pulse harmonic will sport an upper (sum) and lower (difference) sideband. We can see from the geometry of Fig. 14.3(b) that the harmonics (multiples of the sampling rate) must be spaced at least $2 \times f_m$ apart, if the sidebands are not to overlap.

A low-pass filter can be used, as shown in Fig. 14.3(d), to recover the baseband from the pulse train. Realizable filters will pass some of the harmonic bands, albeit in an attenuated form. A close examination of the frequency domain of Fig. 14.3(d) shows a vestige of the first lower sideband appearing in the pass band. However, most of the distortion in the reconstituted analog signal is due to the quantizing error resulting from the crude 3-bit digitization. Such a system will have a S/N ratio of around 20 dB.

In order to reduce the demands of the recovery filter, a sampling frequency somewhat above the Nyquist limit is normally used. This introduces a guard band between sidebands. For example the pulse code telephone network has an analog input bandlimited to 3.4 kHz, but is sampled at 8 kHz. Similarly the audio compact disk uses a sampling rate of 44.1 kHz, for an upper music frequency of 20 kHz.

A more graphic illustration of the effects of sampling at below the Nyquist rate is shown in Fig. 14.4. Here the sampling rate is only 0.75 of the baseband frequency. When the samples are reconstituted by filtering,

[1] Shannon, C.E.; *Communication in the Presence of Noise*, Proc. IRE, vol. 37, Jan. 1949, pp. 10–21.

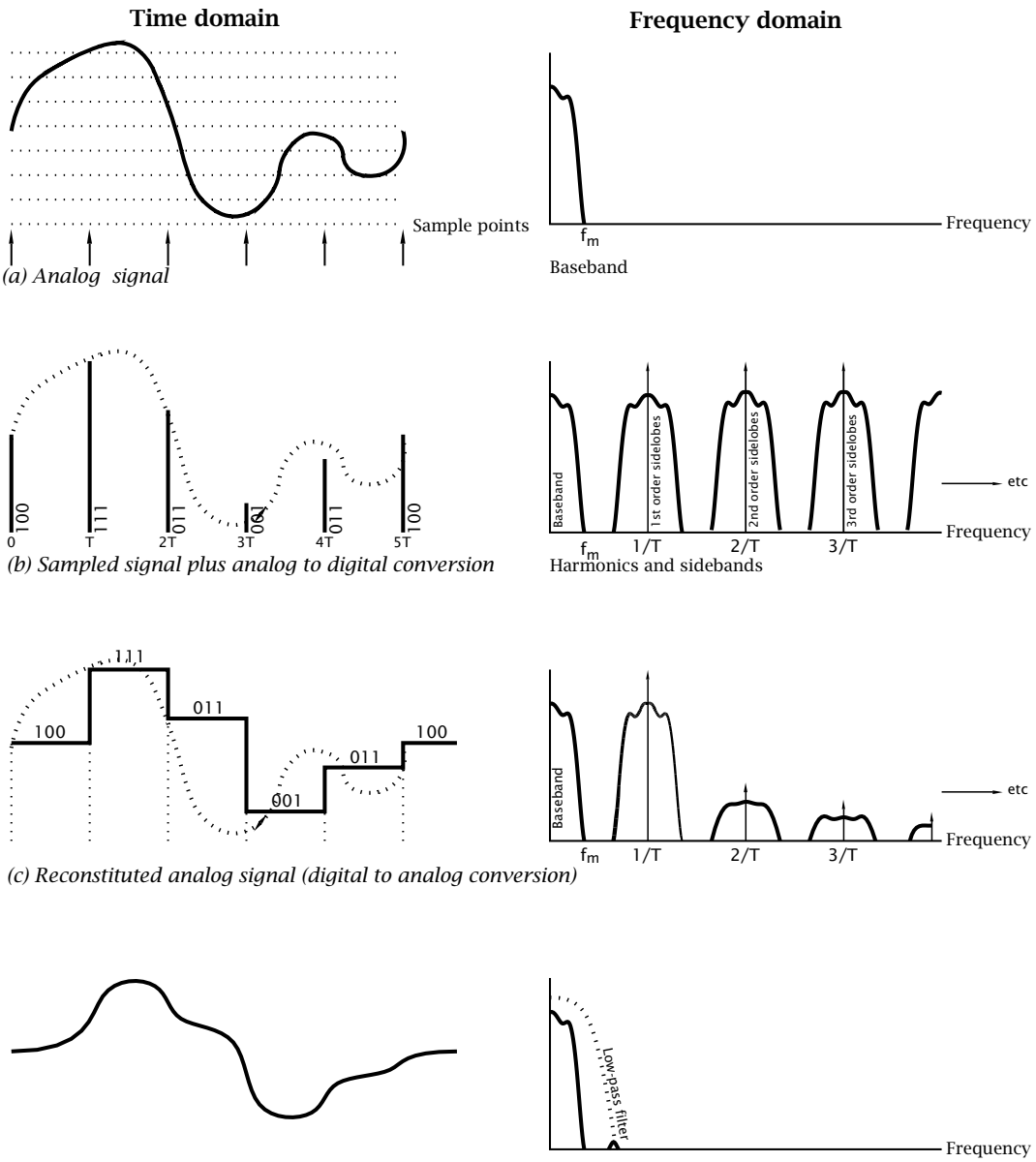
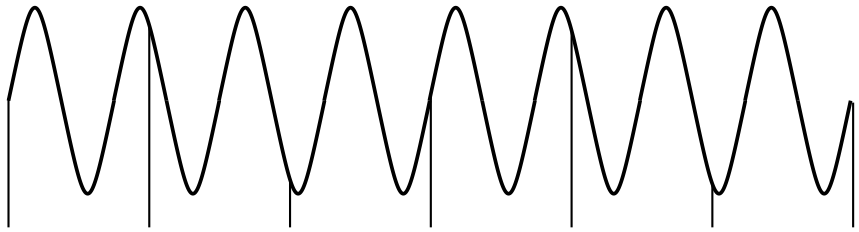


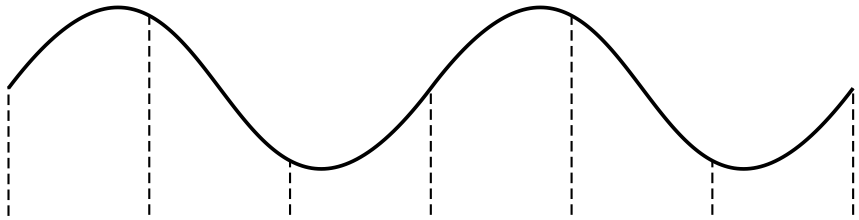
Fig. 14.3 The analog–digital process.

the resulting pulse train, the outcome – shown in Fig. 14.4(b) – bears no simple relationship to the original. This spurious signal is known as an **alias**. Where an input analog signal has frequency components *above* half the sampling rate, maybe due to noise, then this will appear as distortion in the reconstituted signal. For this reason analog signals are usually low-pass filtered at the input of an A/D converter. This process is known as anti-aliasing filtering.

The mapping function from an analog input quantity to its digital equivalent can be expressed as:



(a) Sampling below the Nyquist rate



(b) Resulting filtered signal

Fig. 14.4 Illustrating aliasing.

$$\mathsf{V}_{\mathsf{in}} \mapsto \mathsf{V}_{\mathsf{ref}} \sum_{i=1}^{n} k_i \times 2^{-i}$$

where k_i is the i^{th} binary coefficient having a Boolean value of 0 or 1 and $\mathsf{V}_{\mathsf{in}} \leq \mathsf{V}_{\mathsf{ref}}$ where $\mathsf{V}_{\mathsf{ref}}$ is a fixed analog reference voltage. Thus V_{in} is expressed as a binary fraction of $\mathsf{V}_{\mathsf{ref}}$ and the Boolean coefficients k_{-1} are the required binary digits.

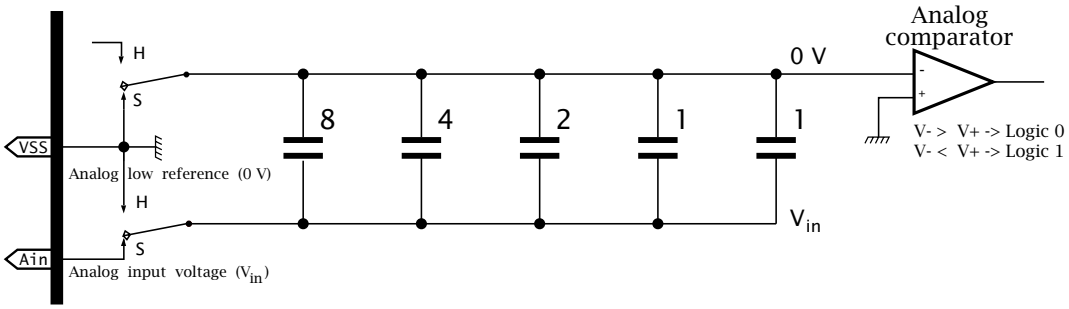
To see how we might implement this in practice, consider the following successive approximation mechanical analogy. Suppose we have an unknown weight W (analogous to V_{in}), a balance scale (compare to an analog comparator) and a set of precision known weights 1, 2, 4 and 8 gm (analogous to an $\mathsf{V}_{\mathsf{ref}}$ of 16 gm). A systemic technique based on the task list might be:

1. Place the 8 g weight on the pan. IF too heavy THEN remove ($k_1 = 0$) ELSE leave ($k_1 = 1$).
2. Place the 4 g weight on the pan. IF too heavy THEN remove ($k_2 = 0$) ELSE leave ($k_2 = 1$).
3. Place the 2 g weight on the pan. IF too heavy THEN remove ($k_3 = 0$) ELSE leave ($k_3 = 1$).
4. Place the 1 g weight on the pan. IF too heavy THEN remove ($k_4 = 0$) ELSE leave ($k_4 = 1$).

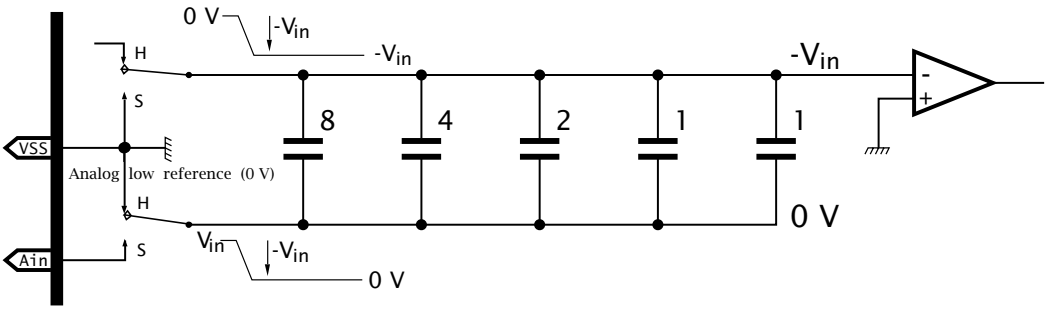
will yield the nearest lower value as the sum of the weights left on the pan. For example if W were 6.2 g then we would have a weight assemblage of 4 + 2 g or $0110b$ for a 4-bit system.

The electronic equivalent to this **successive approximation** technique uses a network of precision resistors or capacitors configured to allow

consecutive halving of a fixed voltage V_{ref} to be switched in to an analog comparator, which acts as the balance scale.



(a) The sample process



(b) The hold process

Fig. 14.5 Initializing the 8-4-2-1 capacitor network.

Most MCUs use a network of capacitors valued in powers of two to subdivide the analog reference voltage, such as shown in Fig. 14.5. Small capacitance values are easily fabricated on a silicon integrated circuit and although the exact value will vary somewhat between different batches of ICs, within the one device this value will closely match and track with changes in temperature and supply voltage. Multiples of the base value can be fabricated by paralleling unit devices – typically gate-source capacitance. The nominal value of a unit capacitor for a PIC16C7XX device is 0.2 pF giving a total capacitance of approximately 52 pF; that is 256×0.2.

Before the conversion process gets underway, the network has to be primed with the unknown analog input voltage V_{in}. This **sampling** acquisition process takes a finite time due to the charging time constant with the resistance of the external circuit together with internal pathways and switch plus a 5 μs analog settling time. If the external resistance is 10 kΩ, and internal resistance approximately 10 kΩ[2], the time constant τ is approximately $50\,\text{pF} \times 20\,\text{k}\Omega = 1\,\mu\text{s}$.

[2]It varies considerable with supply voltage and temperature.

To get to within 0.2% of the final voltage; that is 0.5 of an 8-bit quantum level error, takes approximately $7 \times \tau$. Taken with the 5 μs settling time, the minimum sample time before starting a conversion is around 12 μs. This can be lowered a little by reducing the source resistance. This resistance should not exceed 10 kΩ as pin leakage $I_L = \pm 1\,\mu$A will give a voltage offset approaching the quantum voltage step. Once charged, the sampling switches disconnect the input pin from the network to hold the voltage constant, so that voltage changes during the conversion period do not affect the outcome. Thus in a multi-channel ADC module, the channel selection can be changed at this time.

During the sample (S) period, the top capacitor electrodes are held to 0 V and bottom electrodes are charged to V_{in}. The change-over to the hold (H) position grounds the bottom electrodes and allows the top electrodes to float. The voltage across a capacitor can only change if charge is transferred across electrodes, $\Delta Q = C\Delta V$. Thus the change in voltage $\Delta V = -V_{in}$ at the bottom electrodes is matched at the top floating electrodes, which now become $0 - V_{in}$, as charge cannot flow in or out of the floating top electrodes. Thus at the start of the conversion process the inverting input of the analog comparator is $-V_{in}$.

The successive approximation network at the heart of the A/D converter is shown in a simplified form in Fig. 14.6. The step-by-step process is sequenced by a shift register (SRG, see Fig. 2.20 on page 36) when the programmer sets the GO/$\overline{\text{DONE}}$ bit[3] in the ADCON0 register (A/D CONtrol 0).[4] As the Control shift register is clocked, the single 1 moves down to activate each step in the sequence:

Hold	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Ready	Sample

The capacitor network is switched to Hold and each capacitor, beginning with the largest value, is switched to V_{ref} in turn. The outcome of the comparator then determines the state of the corresponding bit in the Successive Approximation Register (SAR). The process is detailed in Fig. 14.7. After eight set-try-reset actions, the outcome in the SAR is transferred to the Analog to Digital RESult (ADRES) register in File 0A*h*. The GO/$\overline{\text{DONE}}$ flag is now cleared to indicate the End Of Conversion and the ADIF flag set. Finally, the analog input is again switched back into the capacitor network (Sample).

The total conversion time is approximately ten times the clocking rate t_{AD} of the sequencer shift register. The minimum clocking period is 1.6 μs ($\approx$ 600 kHz) for all but the older 2 μs PIC16C71/711 devices. There is no specified lower clocking frequency, but as charge slowly leaks away from the network capacitors, a t_{AD} of more than nominally 20 μs (50 kHz)

[3] GO/$\overline{\text{DONE}}$ can also be set by the CCP2 Comparison special event, see page 375.

[4] The ADCON0/ADCON1/ADRES registers are at File 1F*h*/File 9F*h*/File 1E*h* respectively in all PIC16C7XX devices except the PIC16C71/710/711, where the corresponding locations are File 08*h*/File 88*h*/File 09*h*.

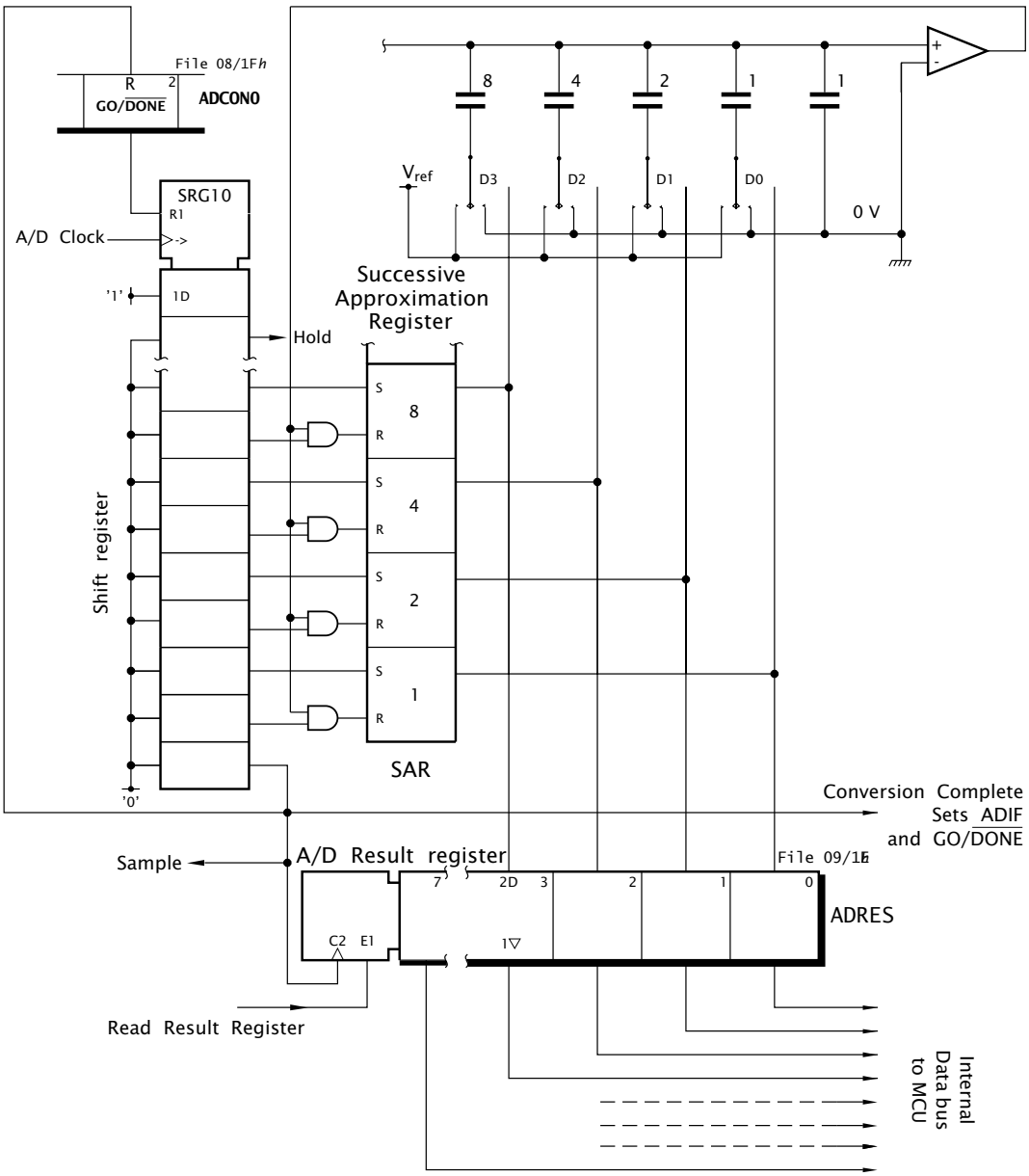


Fig. 14.6 Simplified view of the A/D converter.

should be avoided. From Fig. 14.8 we see that the ADC clock can be derived from one of four sources using the ADCS1:0 (A/D Clock Select) bits in the ADCON0 SPR. The first three of these are fractions of the MCU clock rate and the fourth is a stand-alone CR oscillator with a nominal t_{AD} of 4 μs.

One of the first tasks a programmer must do is to determine the clocking rate by setting the ADCS1:0 bits appropriately. Table 14.2 shows suggested settings for four typical PIC crystal frequencies. If A/D conversion time is critical then the PIC crystal may be chosen to give the fastest con-

Table 14.2: ADC clocking frequency versus device crystal frequency.

ADC clock source t_{AD}		**PIC crystal frequency**				
	ADSC1:0	20 MHz	8 MHz	4 MHz	1 MHz	100 kHz
$f_{osc}/2$	00	—	—	—	2 μs	20 μs
$f_{osc}/8$	01	—	—	2 μs	8 μs	—
$f_{osc}/32$	10	1.6 μs	4 μs	8 μs	—	—
CR	11	2–6 μs	2–6 μs	2–6 μs	2–6 μs	2–6 μs

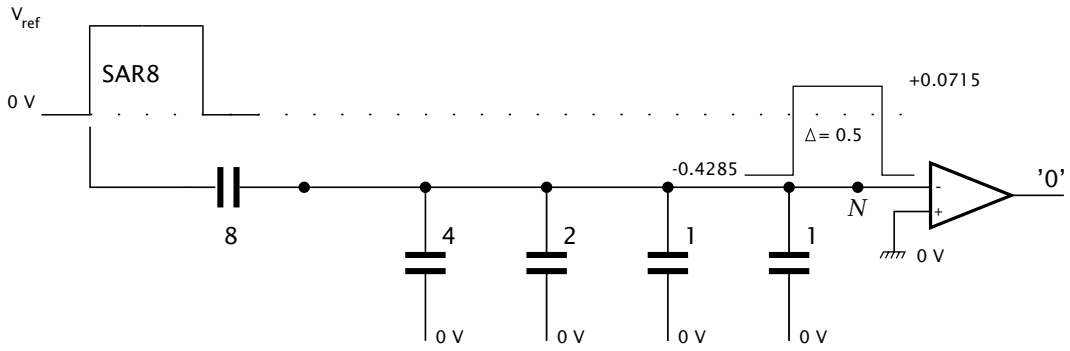
version time. For example, a 5 MHz crystal with ADCS1:0 = 01 gives a t_{AD} of 1.6 μs.

The internal ADC module CR clock is typically used where the main crystal is below 1 MHz. This separate clock source also allows a conversion to be completed when the PIC is in its Sleep mode, as the main processor oscillator is switched off in this situation. In this case the end of conversion interrupt can be used to awaken the MCU. This gives a relatively quiet environment during the conversion and for this reason is often used even where the processor crystal is above 1 MHz – see Program 14.4.

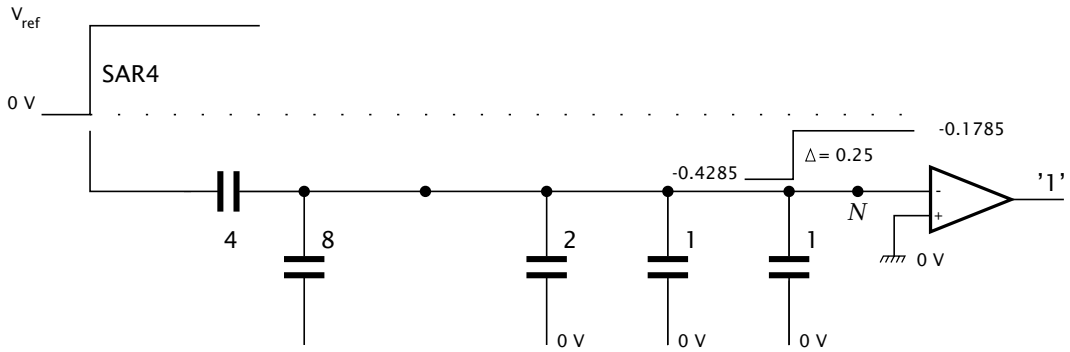
For lowest current consumption, especially during the Sleep mode, the ADC module should be switched off when not in use by clearing the ADON bit in ADCON0. ADON is cleared on Reset, so needs to be set when the module is to be activated.

The conversion process is illustrated in Fig. 14.7. As we have seen in Fig. 14.5, at the end of the sample period the top plates of the capacitor array are at $-V_{in}$ and the bottom plates are disconnected but at zero potential. As an example let us assume that V_{in} is $0.4285V_{ref}$.

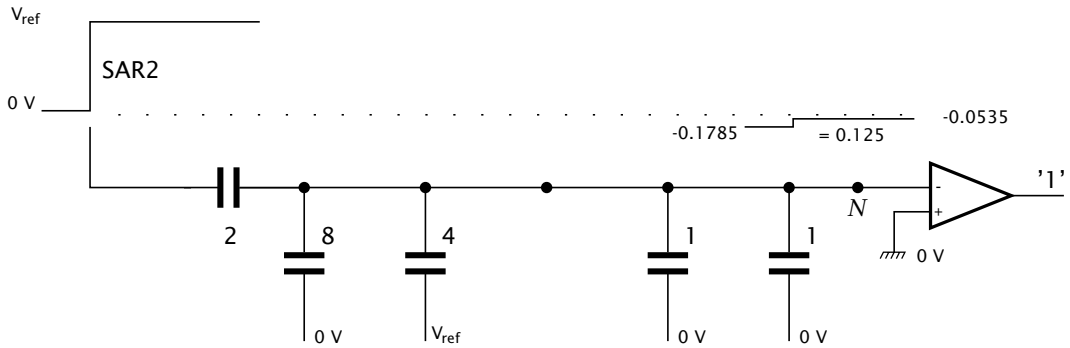
1. The process begins by switching in V_{ref} into the lower plate of the largest capacitor as controlled by the SAR_8 latch in Fig. 14.6. This causes an injection of charge $\Delta Q = C_{total}V_{ref}$, which is identical across both the 8-unit capacitor C_1 and the rest of the capacitors which also have a parallel value of 8 units in Fig. 14.7. Thus the voltage at node N rises by $V_{ref}/2$ to $-0.485 + 0.5 = +0.07125V_{ref}$. In general $\Delta V_N = V_{ref}C_k/C_{total}$. The comparator output is now logic 0 and the SAQ_8 latch is consequently cleared, reversing the $V_{ref}/2$ step.
2. SAQ_4 switches V_{ref} into the next highest capacitor giving a $V_{ref}/4$ step at N ($\frac{4}{12}$). The resulting voltage of $-0.485 + 0.25 = -0.178V_{ref}$ giving a comparator output of logic 1 and SAR_4 remains set with the node voltage staying at $-0.1785V_{ref}$.
3. SAQ_2 switches V_{ref} into the second lowest capacitor giving a $V_{ref}/8$ step at N ($\frac{2}{16}$). The resulting voltage of $-0.1785+0.125 = -0.0535V_{ref}$ giving a comparator output of logic 1 and SAR_2 remains set with the node voltage staying at $-0.0535V_{ref}$.



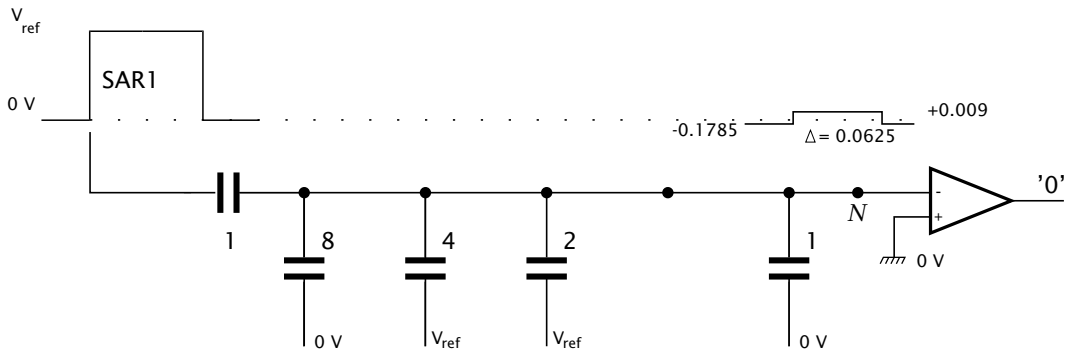
(a) The most significant bit



(b) The second most significant digit



(c) The third most significant digit



(d) The least significant digit

Fig. 14.7 The successive approximation process.

4. SAQ_1 switches V_{ref} into the lowest capacitor giving a $V_{ref}/16$ step at N ($\frac{1}{16}$). The resulting voltage of $-0.0535 + 0.0625 = +0.009V_{ref}$ giving a comparator output of logic 0 and SAR_1 is cleared and the $V_{ref}/16$ step is reversed.

The state of the SAR of $0110b$ or $0.375V_{ref}$ represents the best 4-bit fit to $V_{in} = 0.4285V_{ref}$. The residue $0.0535V_{ref}$ is the quantizing error. Most MCUs use an 8-bit capacitor array. In principle the technique can readily be extended to higher resolutions, but in practice the difficulty in matching ever greater capacitors and internal logic noise means the majority of processors use 8-bit resolution. However, a few MCU devices[5] do have 10 or 12-bit converters. External successive-approximation devices with 12 or more bits resolution, usually using a resistor ladder network, are readily available, but are relatively expensive.

Matching of the array capacitors, offsets and resistance of internal switches, leakage currents and analog comparator non-linearities all contribute to errors in the conversion process. It is beyond the scope of this text to analyze the various measures of error but the device data sheet lists absolute error, defined as the sum of all component error measures, as better than ±1 LSB. This guarantees that the transfer is monotonic; that is the binary code will never move in the reverse direction for any change ΔV_{in} of input voltage. This error figure is for $V_{ref} = V_{DD}$; if V_{ref} is lower than V_{DD} then accuracy deteriorates, although values down to 3 V will give acceptable results in most cases. Accuracy can be improved, especially when the internal CR oscillator is used, if the conversion is done while the PIC is in its Sleep mode.

The standard PIC ADC module has eight input channels with any one selected for conversion according to the 3-bit Channel Select code CHS2:0 in the ADCON0 register, as shown in Fig. 14.8. 28-pin footprint PICs, such as the PIC16C73, can only access the bottom five channels. The PIC16C71 line of 18-pin footprint[6] and 12C67X devices have an earlier 4-channel module with CHS2 missing. The PIC16C774 uses ADCON0[1] as CHS3.

The input analog channels AN4:0 are shared with the Port A digital inputs RA4:0 and AN7:5 with Port E RE2:0 in 40-pin devices. AN3 is special in that it can be used as the reference voltage input if configured accordingly by PCFG0 = 1 (Port ConFiGuration) bit in the ADCON1 register. Like all port configuration registers it is normally set up only once at the beginning of the program and it is therefore located in the less convenient Bank 1 Data memory. Any such low-noise external V_{ref} should be in the range 3 V → V_{DD} + 0.3 V – see Fig. 14.17. For best accuracy it should be as high as possible; a value of 5.12 V will give a 20 mV per bit resolution.

[5] For example the PIC16F87X devices have a 10-bit ADC and the PIC16C77X have a 12-bit ADC with internal precision positive and negative reference voltages.

[6] The PIC16C71/710/711/715.

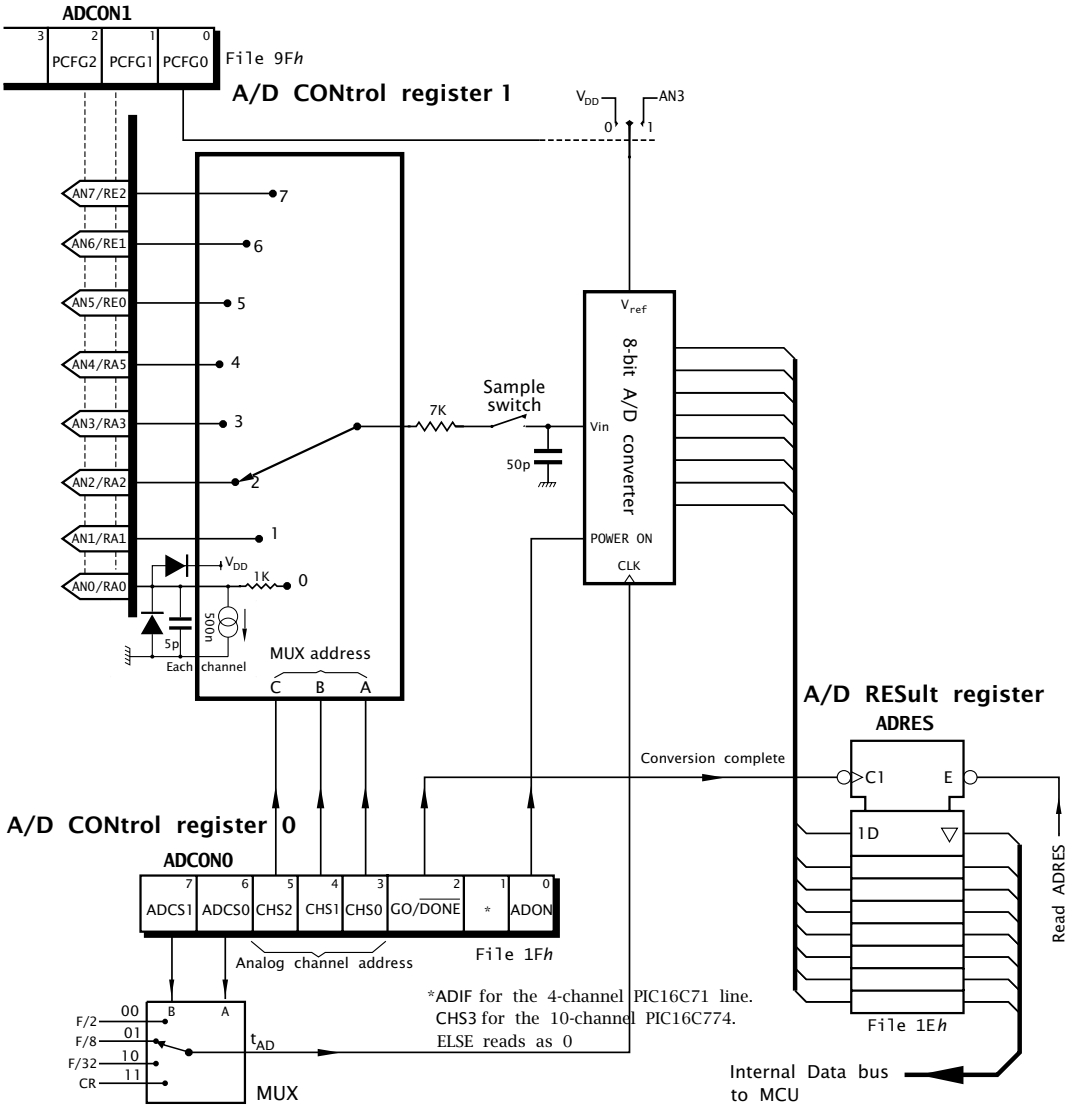


Fig. 14.8 The 8-bit 8-channel analog to digital conversion module.

A low cost option is to use the standard supply voltage V_{DD} (PCFG0 = 0) as the reference voltage, thus freeing up AN3 for use as a normal analog input. To reduce noise, a 0.1–1 μF Tantalum electroytic capacitor should be used to decouple noise as close as possible to the V_{DD} pin.

If less than eight analog channels are needed then some of the pins can be assigned as digital I/O port lines using the PCFG2:0 bits as listed in Table 14.3. For example, if PCFG2:0 = 100 then RA1:0 are both analog (AN1:0) and the rest are digital (RA4:2, RE2:0) as applicable, with V_{DD} used as the reference.

On Reset all pins are set to accept analog signals (A). Pins that are reconfigured as digital I/O (D) should never be connected to an analog signal. Such voltages may bias the digital input buffer (see Fig. 11.2 on

Table 14.3: Configuring the ADC port pins in the PIC16C73/74 devices.

PCFG2:0	AN7 RE2	AN6 RE1	AN5 RE0	AN4 RA5	AN3 RA3	AN2 RA2	AN1 RA1	AN0 RA0
000	A	A	A	A	A	A	A	A
001	A	A	A	A	V_{ref}	A	A	A
010	D	D	D	A	A	A	A	A
011	D	D	D	A	V_{ref}	A	A	A
100	D	D	D	D	A	D	A	A
101	D	D	D	D	V_{ref}	D	A	A
110	D	D	D	D	D	D	D	D
111	D	D	D	D	D	D	D	D

page 273) into its linear range and the resulting large current could cause *irreversible* damage.

Other PIC devices with an ADC module may have different settings and numbers of PCFG bits. For example the 8-pin footprint PIC12C67X devices with a 4-channel ADC module can configure individual pins as analog or digital to best use scarce resources. The PIC16C71X line has only the two PCFG bits whilst the PIC16C774 has four.

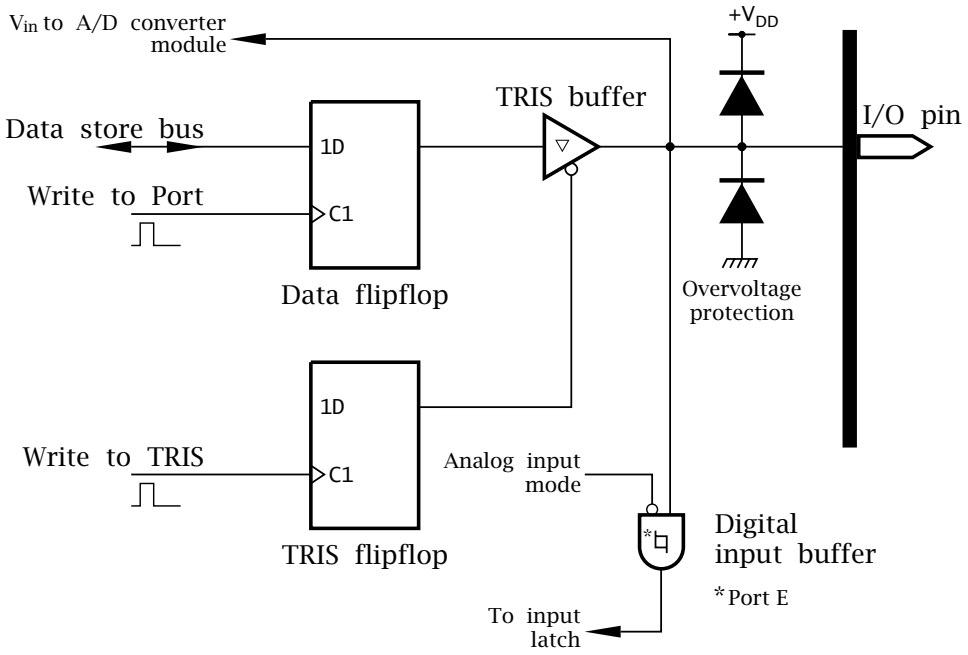


Fig. 14.9 Configuring the analog inputs for Port A and Port E.

We can see from Fig. 14.9 that an I/O pin configured as an analog input from ADCON1 simply disables the digital input buffer (compare with Fig. 11.2 on page 273). No other circuitry is affected. From this we can make the following deductions.

- A port pin configured as analog will read as logic 0 due to the disabled digital input buffer.
- The TRIS buffer is not affected and thus the appropriate TRIS bits should be 1; that is the direction of the port pins configured as analog should be set to input to prevent contention between the nalog V_{in} and the digital state of the Data flip flop.
- The ADC can read an analog voltage at the pin even if that pin has not been configured as analog. However, the still active digital input buffer may consume an excessive current outside of the device's specification.

Using Fig. 14.8 as the programmer's model we can now deduce the hardware-software interaction in order to action a conversion. Assuming first that interrupts are not being used, the following steps can be identified:

1. Configure ADC module.
   - Set up port pins as analog/voltage reference (ADCON1).
   - Select ADC conversion clock source (ADCON0).
   - Select ADC input channel (ADCON0).
   - Turn on ADC module (ADCON0).
2. Wait for the required acquisition time, typically 12 μs.
3. Start conversion by setting the GO/$\overline{\text{DONE}}$ bit.
4. Wait for ADC conversion to complete by polling the GO/$\overline{\text{DONE}}$ bit for low.
5. Read the ADRES register.
6. For next conversion go to step 1 or step 2 as required.

As an example, consider that we wish to read the channel n analog voltage (RAn) of a PIC16C74 and output the equivalent digital value at Port B. The main crystal is 20 MHz and V_{DD} is to be used as V_{ref}.

The listing of Program 14.1 assumes that the ADC module has been initialized at reset with startup code of the form:

```
    bsf    STATUS,RP0     ; Bank 1
    clrf   ADCON1         ; All port inputs are analog
    bcf    STATUS,RP0     ; Back to Bank 0
    movlw  b'10000001'    ; AD clock/32, Ch0, no convert, ADON
    movwf  ADCON0
```

which sets up the pin configuration according to the PCFGn settings of Table 14.3 to enable all eight ADC channels. The ADCON0 is initialized to $\underline{10}\,\underline{000}\,\underline{0}\underline{0}\underline{1}b$ to set the module clock source as the crystal frequency/32; i.e. $\frac{20}{32} = 625$ kHz (giving a conversion time of $\approx 12 + 15\mu$s), channel zero and the module is turned on. With an initial zero value of GO/$\overline{\text{DONE}}$ no conversion is actioned. The initial channel value is irrelevant.

With the module initialized, the subroutine listed in Program 14.1 simply copies the contents of W, truncated to three bits for robustness, into a temporary location TEMP. There it is logic shifted left three places to align the channel number with the CHSn bits in the ADCON0 register.

Program 14.1 Taking a reading from channel n.

```
; ********************************************************
; * FUNCTION: Analog/digital conversion at channel n     *
; * RESOURCE: Subroutine DELAY_12US, byte TEMP           *
; * ENTRY   : Channel number in W                        *
; * EXIT    : Digitized analog value in W                *
; ********************************************************
GET_ANALOG
        andlw  b'0111'        ; Isolate address bits
        movwf   TEMP          ; Channel number
        bcf    STATUS,C       ; Shift channel number left >>3
        rlf    TEMP,f
        rlf    TEMP,f
        rlf    TEMP,w
        bcf    ADCON0,CHS0    ; Zero channel bits
        bcf    ADCON0,CHS1
        bcf    ADCON0,CHS2
        addwf  ADCON0,f
        call   DELAY_12US     ; Wait 12us to stabilize
        bsf    ADCON0,GO      ; Start conversion
GET_ANALOG_LOOP               ; Takes around 15us to finish
        btfsc  ADCON0,NOT_DONE; Check for End Of Conversion
         goto  GET_ANALOG_LOOP
        movf   ADRES,w        ; Fetch when GO/NOT_DONE zero
        return


; ********************************************************
; * FUNCTION: Delays 12us at 20MHz                       *
; * ENTRY   : None                                       *
; * RESOURCE: None                                       *
; * EXIT    : W is zero                                  *
; ********************************************************
DELAY_12US
        movlw  d'14'           ; Delay constant
DELAY_12US_LOOP
        addlw  -1              ; Decrement
        btfss  STATUS,Z        ; Until zero
         goto  DELAY_12US_LOOP
        return
```

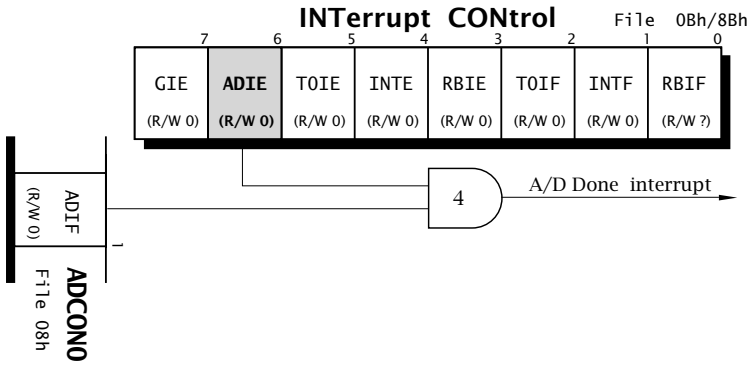
After clearing the CHS2:0 bits, the shifted channel number can then be added into ADCON0 to set CHS2:0 to the appropriate channel.

Once the channel number has been set up, a 12 μs delay subroutine is called and then the GO/$\overline{\text{DONE}}$ bit in ADCON0 is set to initiate a conversion.[7] The completion of the process can then be monitored by polling GO/$\overline{\text{DONE}}$ until this goes low. Notice that the p16c74.inc include file for clarity allows the programmer to use either bit name GO or NOT_DONE

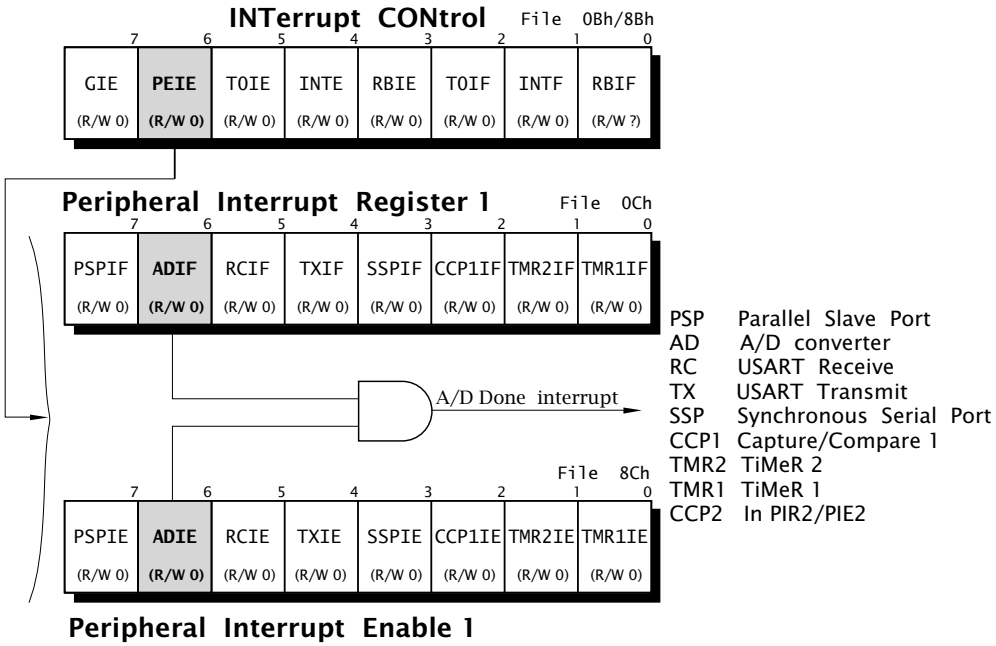
[7] A conversion may be aborted at any time by clearing GO/$\overline{\text{DONE}}$.

synonymously for bit 2. At this point the content of ADRES is the result of the conversion.

Rather than polling for completion, the end of conversion can be used to generate an interrupt. In particular if a conversion is to be done in the Sleep mode then this interrupt can be used to awaken the device.



(a) PIC16C71 interrupt control



(b) PIC16C73/4 interrupt control

Fig. 14.10 Interrupt control for the ADC module.

The procedure in obtaining a digitized outcome using the ADC interrupt is similar to the polling technique.

1. Configure ADC module (see Table 14.3).
2. Configure ADC interrupt.
   - Set the ADIE local mask bit to enable the ADC module interrupt.

   - Clear the ADIF flag bit.
   - Set the PEIE auxillary peripheral interrupt group mask bit (not the PIC16C71X line).
   - Set the GIE bit to globally enable all interrupts.
3. Wait for the required acquisition time, typically 12 μs.
4. Start conversion by setting the GO/$\overline{\text{DONE}}$ bit.
5. Continue .....until interrupted.
6. Interrupt service routine (ISR) checks ADIF bit and if set reads the ADRES register and clears the ADIF bit.
7. For next conversion go to Step 2 or 3 as required.

To illustrate the process consider an interrupt-driven equivalent to the subroutine of Program 14.1, where we wish to action the digitization of the analog voltage at channel n. There are three considerations for this situation.

- Initialization code.
- The conversion subroutine.
- The interrupt service routine.

The initialization code depends a little on the target PIC device. The 18-pin footprint PIC16C71X line has a 4-channel ADC module as the counterpart to the PIC16F83/4's Data EEPROM module. This can be seen by comparing the INTerrupt CONtrol register of the latter in Fig. 14.10(a) with the former shown in Fig. 7.4 on page 178. Here the ADC module's Interrupt Enable mask bit ADIE replaces the EEIE mask as INTCON[6] and the ADC module's Interrupt Flag is located as ADCON0[1] (see Fig. 14.8).

The PIC16C73/4 28/40-pin line has a much larger line-up of peripheral devices and use two pairs of additional registers to service these auxillary[8] interrupt mask and flag bits. These Interrupt registers are called the Peripheral Interrupt and Peripheral Interrupt Enable registers. PIR1 and PIE1 are shown in Fig. 14.10(b). The former at File 0Ch holds eight interrupt flags, including ADIF at PIR1[6]. The latter at File 8Ch holds the corresponding mask bits. There is also a PIR2/PIE2 pair at File 0Dh:8Dh. Both these register pairs are enabled as a group by the PEIE (PEripheral Interrupt Enable) bit which takes the place of the single mask bit in INTCON[6] in the 18-pin footprint devices. Thus in enabling the ADC module for interrupt handling, the programmer has to enable both the ADIE local and the PEIE group masks.

In our example, we assume a PIC16C74 target device. Then our initialization code might be:

[8] That is, apart from the standard external INT, Port B Change and Timer 0 interrupt sources.

```
bsf   STATUS,RP0  ; Bank 1
clrf  ADCON1      ; All port inputs are analog
bsf   PIE1,ADIE   ; Enable ADC local interrupt (not PIC16C71)
bcf   STATUS,RP0  ; Back to Bank 0
movlw b'10000001' ; AD clock /32, Ch0, no convert, ADON
movwf ADCON0
bcf   PIR1,ADIF   ; ADCON0,ADIF for PIC16C71X
bsf   INTCON,PEIE ; INTCON,ADIE for PIC16C71X
bsf   INTCON,GIE  ; Enable interrupt subsystem
```

The interrupt-driven version of our subroutine shown in Program 14.2 is virtually identical to Program 14.1 except that the polling loop at the exit point is eliminated and, of course, no value is returned. However, the 12 μs delay before commencing the process is still needed.

The ISR shown in Program 14.3 is entered when an interrupt (from any source) is generated. For simplicity we assume that there are no other sources of interrupt except from the ADC module. If this is the case, technically the check of the ADIF flag is redundant, although a spurious interrupt can never be ruled out. Where there are multiple sources of interrupt, then each flag can be tested in turn as shown on page 179.

In Program 14.3 a datum file register NEW is cleared to show the background program that the datum byte in ADRES has never been read. When the background routine fetches the digitized byte in ADRES it sets the flag byte NEW to a non-zero value. In a more sophisticated system a buffer of several digitized samples can he maintained by the ISR with NEW giving the number of samples in the buffer – see Example 14.1.

Program 14.2 Interrupt-driven subroutine to read channel n.

```
; ********************************************************
; * FUNCTION: Analog/digital conversion at channel n     *
; * RESOURCE: Subroutine DELAY_12US, byte TEMP           *
; * ENTRY   : Channel number in W                        *
; * EXIT    : Conversion initiated                       *
; ********************************************************
GET_ANALOG
        andlw  b'0111'      ; Isolate address bits
        movwf   TEMP        ; Channel number
        bcf    STATUS,C     ; Shift channel number left >>3
        rlf    TEMP,f
        rlf    TEMP,f
        rlf    TEMP,w       ; with outcome in W
        bcf    ADCON0,CHS0  ; Zero channel bits
        bcf    ADCON0,CHS1
        bcf    ADCON0,CHS2
        addwf  ADCON0,f
        call   DELAY_12US   ; Wait 12us for things to stabilize
        bsf    ADCON0,GO    ; Start conversion
        return
```

Program 14.3 The ISR for our interrupt-driven ADC software.

```
; ********************************************************
; * FUNCTION: ISR to read the ADC module at EOC          *
; * ENTRY   : On an interrupt                            *
; * EXIT    : Set NEW to zero to show new value in ADRES *
; ********************************************************
; First save context
A_D_ISR  movwf   _work     ; Put away W
         swapf   STATUS,w  ; and the Status register
         movwf   _status

; ************************************************************

         btfss   PIR1,ADIF ; Check; has there been a conversion
          goto   ISR_EXIT  ; IF not THEN false alarm
         clrf    NEW       ; Show outside world a new outcome
         bsf     PIR1,ADIF

; ************************************************************

ISR_EXIT swapf   _status,w ; Untwist the original Status reg
         movwf   STATUS
         swapf   _work,f   ; Get the original W reg back
         swapf   _work,w   ; leaving STATUS unchanged
         retfie            ; and return from interrupt
```

The core of the ISR is sandwiched by code to save and restore the Working and Status registers as described in Program 7.2 on page 183. As the PIR1 register is available in both Banks 0 & 1, the more complex context saving code illustated in Program 7.4 on page 191 is not required.

The ADC module can operate when the PIC is in its Sleep state. Indeed, a conversion during the (electrical) peace and quiet of sleep may well be preferrable than normal operation. When the ADC internal CR clock is used to sequence the analog module because the system clock is too low, then Sleep conversion is recommended as the CR and system clock are not synchronized and clock feed-through noise is a problem.

The following task list outlines the Sleep state conversion process.

- The ADC clock source must be set to CR, ADCS1:0 = 11.
- The ADIF flag must be cleared to prevent an immediate interrupt.
- The ADIE mask bit must be set to enable the ADC interrupt to awaken the processor.
- The GIE mask bit mask bit must be 0 unless the programmer wishes the processor to jump to an ISR when it awakens.
- The GO/DONE bit in the ADCON0 register must be cleared to initialize the conversion followed immediately by the `sleep` instruction.
- On awakening, the ADRES holds the digitized value.

Program 14.4 shows a Sleep state conversion, assuming that initialization code similar to the following has been executed.

```
    include "p16c71.inc"
    bsf     STATUS,RP0 ; Bank 1
    movlw   b'010'     ; RA1:0   set to AN1:0 analog with
    movwf   ADCON1     ; PCFG1:0 = 10; rest of PortA digital
    bcf     STATUS,RP0 ; Back to Bank 0
    movlw   b'11000001'; RC clock, Ch0, no convert, ADIF = 0
    movwf   ADCON0     ; and ADON
    bcf     INTCON,ADIE; Enable ADC interrupt in 16C71
    bcf     INTCON,GIE ; Disable interrupt subsystem
```

This code shows the PIC16C71 as the target processor with its two PFCG1:0 bits set up to configure RA1:0 as analog inputs. The internal CR ADC module's oscillator has been chosen as the clock source and ADC Interrupt flag ADIF (in the iNTCON register for the PIC16C71X line) cleared. Setting ADIE enables the ADC module's interrupt system. Where applicable the PEIE mask bit at INTCON[6] must also be set. If these are not enabled, then entering the Sleep state will turn off the ADC module in the normal Sleep manner, although the ADON bit reamins 1, and the conversion is aborted. This will also occur if the CR clock option was not choosen.

Program 14.4 Digitizing Channel 1 of a PIC16C71 device.

```
NEW_AD  bsf   ADCON0,CHS0   ; Select Channel 1
        bcf   ADCON1,CHS1   ; for conversion
        call  DELAY_12US    ; Wait for things to settle
        bsf   ADCON0,GO     ; Start conversion
        sleep               ; Go to sleep
; When A/D is over, program will continue here if GIE is 0
        bcf   ADCON0,ADIF   ; Clear interrupt flag and
        movf  ADRES,w       ; go get the value
        return
```

Program 14.4 shows a conversion being implemented from Channel 1 (CHS1:0 = 01). After the channel has been set up and time allowed for settling[9] the GO/$\overline{\text{DONE}}$ bit is cleared to start the conversion process. Where the CR oscillator option has been choosen, there is a 1-instruction cycle delay inserted to allow for the following sleep instruction to close down the processor.

After the PIC has awakened, the ADIF can be cleared and the digitized value read from the ADRES register.

As was the case for the other peripheral devices described in earlier chapters, **C** code may be used to interact with the ADC module. The

[9]If the main oscillator is low frequency, one or two nop instructions may be all that is needed. If the channel has not been changed and sufficient time has elapsed since the last conversion, then no extra delay is needed.

various configuration ports may be accessed in the same manner as at assembly level or, as in Program 14.5, the appropriate compiler built-in functions used to set up and manipulate the SPR control and status bits.

For our example we are going to code a 20 MHz PIC16C74 to act as a comparator in the manner of the Example 11.2 on page 291. Here we want to compare the parallel-input 8-bit word N at Port B with the analog input at Channel 1. Outputs at RC2:0 are to represent Analog Lower Than N, Equivalent and Higher Than N respectively. The comparator is to have a hysteresis of ± 1 bit. That is, if previous comparisons showed Analog $< N$ then the trigger level is $N + 1$ for equality. Similarily, on a downward trajectory the trigger level is decreased to $N - 1$ for equality.

The function compare() of Program 14.5 assumes that initialization code of the form:

```
#include <16c74.h>
#use     delay(clock=20000000)
#define  PORT_B *(unsigned int *)0x06
#define  PORT_C *(unsigned int *)0x07

void compare(unsigned int delta);
int main()
{
unsigned int hysteresis = 0;
set_tris_c(0xF8);
setup_adc(ADC_CLOCK_DIV_32);
setup_adc_ports(RA0_RA1_RA3_ANALOG);
set_adc_channel(1);
delay_us(12);
```

has already been executed.

The key internal functions used here are:

setup_adc(ADC_CLOCK_DIV_32);
This function configures bits ADCS1:0 in the ADCON0 to select the module's clock source; here the processor oscillator/32. The script ADC_CLOCK_INTERNAL may be used to select the internal CR oscillator.

setup_adc_ports(RA0_RA1_RA3_ANALOG);
This configures bits PCFG2:0 in ADCON1 to select which port pins are analog, which are digital and if an external V_{ref} is to be used. The script RA0_RA1_RA3_ANALOG indicates that port lines RA3 and RA1:0 are to be analog with an internal V_{ref} with the rest being digital (PCFG2:0 = 100, see Table 14.3. The equivalent script using an external V_{ref} at RA3 is RA0_RA1_ANALOG_RA3_REF. Scripts appropriate to any aprticular device are stored in the corresponding header file, in this case 16c74.h. All devices with an analog module have scripts ALL_ANALOG and NO_ANALOGS.

set_adc_channel(n);
Used to set up the channel number n in ADCON0, bits CHS2:0.

`read_adc();`
Activates GO/$\overline{\text{DONE}}$ in ADCON0 and returns with the digitized value from ADRES when GO/$\overline{\text{DONE}}$ goes to 1.

`delay_us(n);`
Not explicitely an analog module function; this delays by $n\,\mu$s. The `#use delay(clock=20000000)` declaration is used by this function to give the delay appropriate for the processor clock frequency.

If the channel number remains constant and the `compare()` function is not called up at intervals less than the $12\,\mu$s settling time (the normal situation) then `delay_us(12);` can be omitted.

Program 14.5 A digital/analog comparator with hysteresis.

```
void compare(unsigned int delta)
{
unsigned int analog;
analog = read_adc();
if(analog > PORT_B + delta) {PORT_C = 0x04; delta = 0xff;}
if(analog == PORT_B) {PORT_C = 0x02;}
else {PORT_C = 0x01; delta = 1;}
return delta;
}
```

The function `compare()` in Program 14.5 expects the value of the hysteresis, which here is either +1 or −1 (FFh). Internally this is called `delta`. After the ADC module is read, the digitized value `analog` is compared with the contents of Port B plus `delta` and the three Port C bits (`RC2:0`) set to their appropriate state.

At the same time as the comparison is resolved, `delta` will be updated to reflect the outcome (i.e. +1 if `analog` < (`PORT_B` + `delta`), −1 if `analog` > (`PORT_B` + `delta`)). The value `delta` is returned by the function to allow the caller function to update its variable `hysteresis`. Thus to activate the comparator outputs and also update `hysteresis` at the same time the caller might have a statement such as `hysteresis = compare(hysteresis);`. An alternative would be to define the variable `hysteresis` before the main function `main()` making it global; that is known to all functions. In this situation its value need not be passed by the caller back and forth to any appropriate function.

The declarations `#use fast_io(n);` (see page 296) have not been used here as the `input(pin)` and `output_bit(pin,value)` internal functions have not been used and the ports have been treated as simple memory bytes.

Conversion from a digital quantity to an analog equivalent is somewhat simpler than the converse and not so commonly required. Perhaps

for these reasons digital to analog converters (DACs) are not often found as an integral function in most MCU families.

We have already seen that a rather crude way of providing this mapping is to vary the mark:space ratio of a pulse train of constant repetitive duration, as shown in Fig. 13.9 on page 380. Here a small digital number gives a skinny pulse, which when smoothed out by a low-pass filter (which gives the average or d.c. value) translates to a low voltage. Conversely, a large digital number leads to a correspondingly large mark:space ratio, which in turn after smoothing yields a higher voltage.

PWM conversion can be very accurate and is simple to implement. However, extensive filtering is required to remove harmonics of the pulse rate and this makes the conversion slow to respond to changes in the digital input. Normally PWM is used to control heavy loads, such as motors or heaters, where the inertia of these devices inherently provides the smoothing action. Furthermore, the pulsed nature of the signal is ideally suited to power control, activating thyristor firing circuits.

Many commercial DAC devices are available which can be controlled via standard digital I/O ports. Two examples were given in Figs. 12.4 and 12.6 on pages 310 and 314 where the MCU transferred digital data in series. Here we will look at an example where parallel data transfer is used.

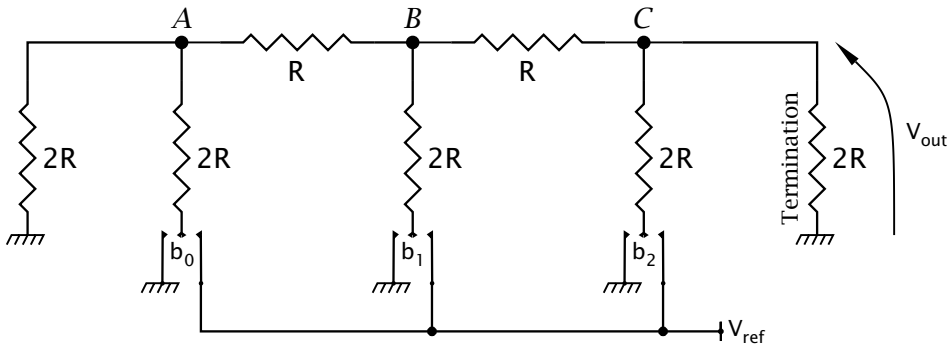
The majority of proprietary devices are based on an R-2R ladder network, such as that shown in Fig. 14.12(a). Voltage appearing at any bit switch node emerges at the output node in an attenuated form. As our analysis will show, each move to the left attenuates this voltage b_n by 50%, which is the binary weighting relationship:

$$V = \sum_{i=0}^{N+1} b_n \times 2^i$$

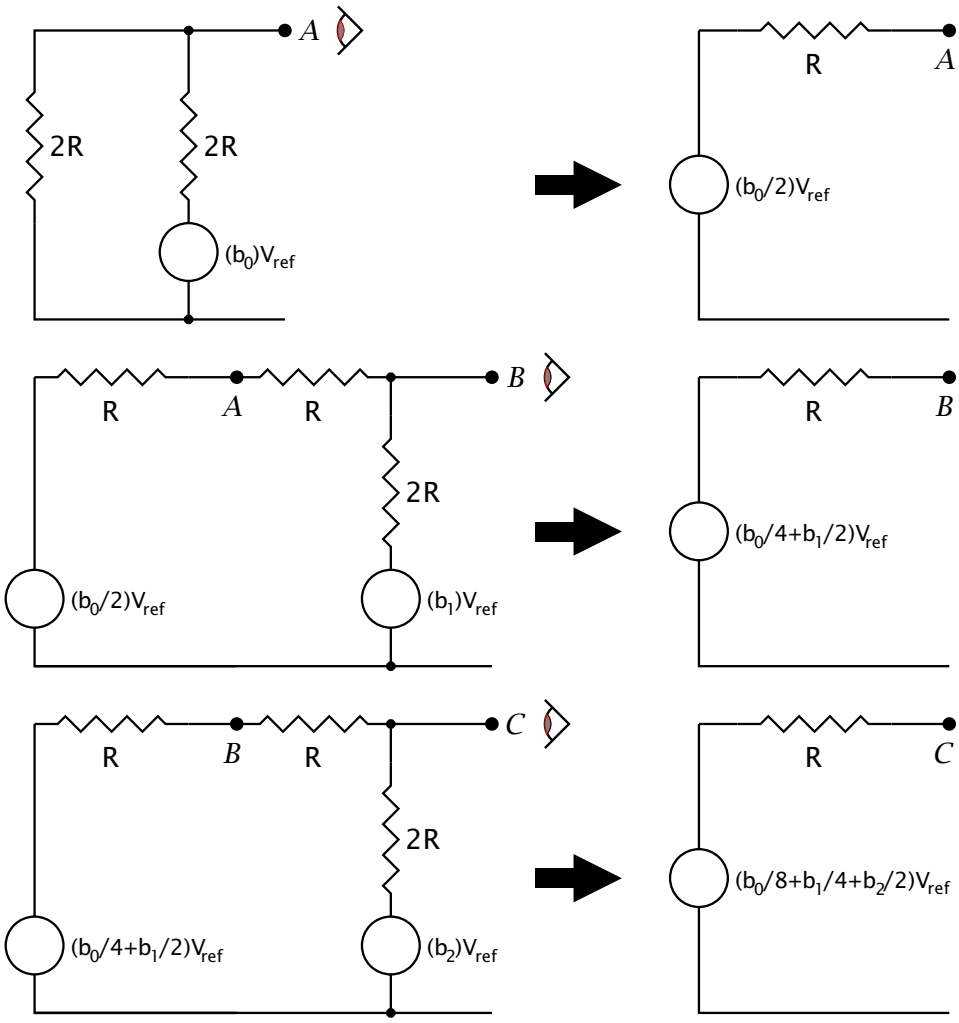
for an N-bit word.

In Fig. 14.6(b) at mode A looking to the left we see a resistance of R ($2R//2R$) and the voltage is attenuated by two. As we move to the right the process is repeated with each voltage divided by two. Thus, at node B the voltage $b_0/2$ is further divided by two and the next digital node voltage is divided by two giving $V_B = b_0/4 + b_1/2$. As the network is symmetrical the resistance looking right at any mode is also $2R$. This means that as seen from *any* digital switch, the total resistance is $2R + 2R//2R = 3R$. This is important as the characteristics of a transistor switch, such as resistance, are dependent on current and keeping this the same reduces error.

For clarity our analysis has been for three bits. This can be extended by simply moving the leftmost terminating resistor over and inserting the requisite number of sections. This does not affect the resistance as seen left of the mode, and therefore does not change the conditions of the



(a) A 3-bit R-2R ladder network



(b) Reducing the circuit

Fig. 14.11 R-2R digital-to-analog conversion.

rightmost sections. An inspection of our analysis shows that nowhere does the absolute value of resistance appear. In fact the accuracy of the analysis depends only on the R:2R ratio. While it is relatively easy to fabricate accurate ratioed resistors on a silicon die, this is certainly not the case for absolute values. For this reason R:2R networks are the standard technique used for most integrated circuit DACs.

The Maxim MAX506 is an example of a commercial D/A converter (DAC). This 20-pin footprint device contains four separate DACs sharing a common external V_{ref}. Digital data is presented to the D7:0 pins and one of four latch registers selected with the A1:0 address inputs. Once this is done, the datum byte is loaded into the selected register n and appears at the corresponding output VOUTn.

This output analog voltage ranges from zero (Analog GrouND) for a digital input of 00h through to V_{ref} for a digital input of FFh.

Where V_{SS} is connected to ground then V_{ref} can be anything between 0 V and V_{DD} (+5 V). However, V_{SS} can be as low as −5 V and in this case V_{ref} can be anywhere in the range ±5 V. If V_{ref} is negative for dual supplies then the output voltage will also be negative. In either case, effectively the output can be treated as the product $D \times V_{ref}$ where D is the digital input byte scaled to the range 0 – 1 (00 – FFh).

The MAX505 24-pin variant allows for separate reference voltages to be used for each of the four DAC channels. In addition, the MAX505's DAC latches are isolated from the converter ladder circuits by a further layer of latches all clocked at the *same* time with a $\overline{\text{LDAC}}$ ($\overline{\text{Load DAC}}$) control signal. This double buffering permits the programmer to update all four DACs simultaneously after their individual latches have been set up.

As an example, consider that a MAX506 quad DAC has its Address selected via RA1:0 and RA2 drives the $\overline{\text{WR}}$ input to latch in the addressed data from Port B. A software routine to generate a continuous staircase sawtooth waveform from DACD would look something like:

```
      movlw   b'0111'      ; DACD is channel 3, WR = 1
      movwf   PORTA        ; To MAX506 WR, A1:0
LOOP  movwf   PORTB        ; Datum to MAX506's D7:0
      bcf     PORTA,2      ; WR = 0; Latch datum in
      bsf     PORTA,2      ; WR = 1; by pulsing WR
      addlw   1            ; Increment staircase count
      goto    LOOP         ; and repeat forever
```

where we are assuming that Port B and Port A[2:0] have been set up as outputs.

A typical DAC staircase output waveform is shown in the oscillogram in Fig. 14.13. Here a 12 MHz crystal clocked PIC is shown which, with a

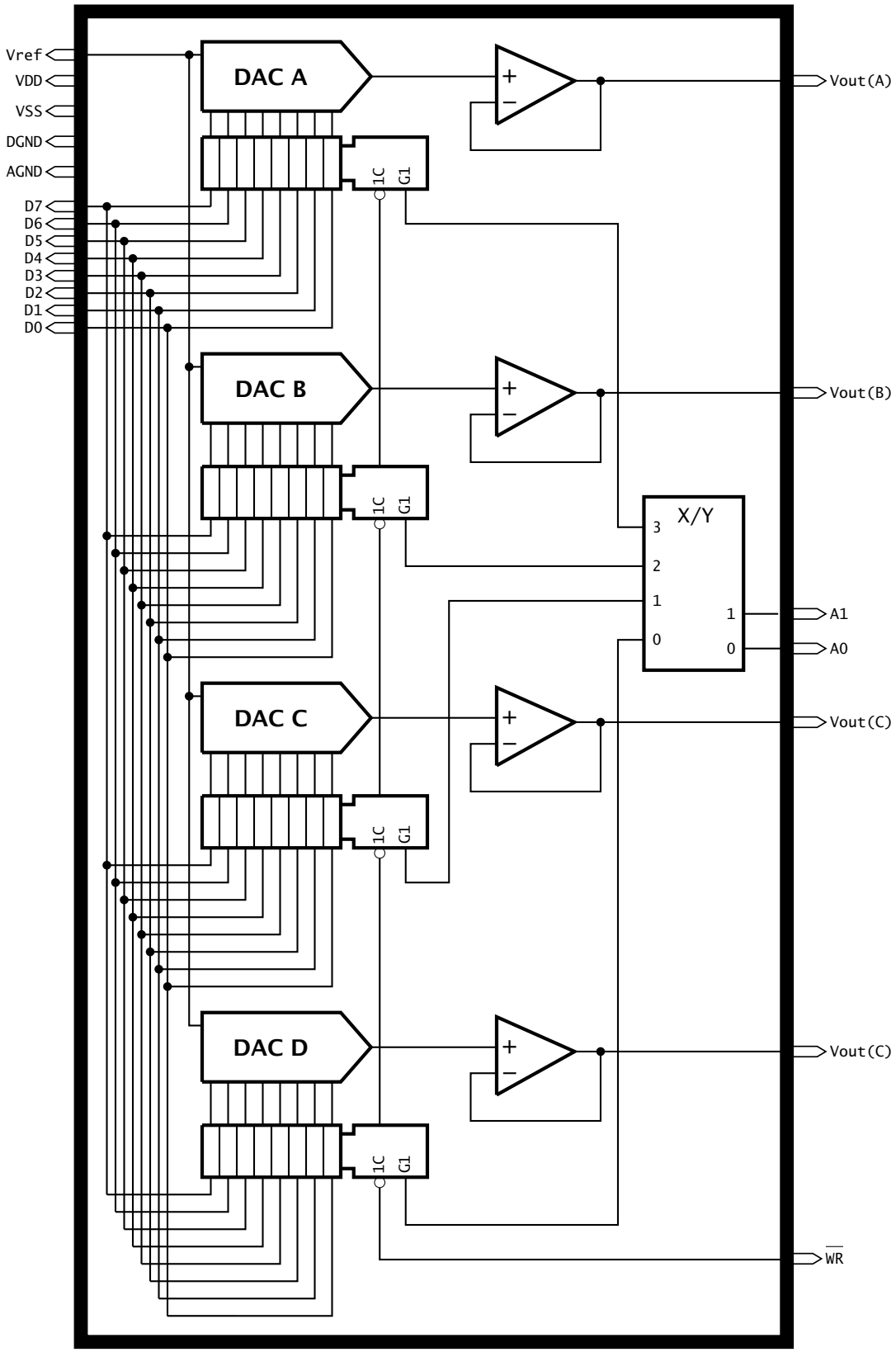


Fig. 14.12 The Maxim MAX506 quad 8-bit D/A converter.

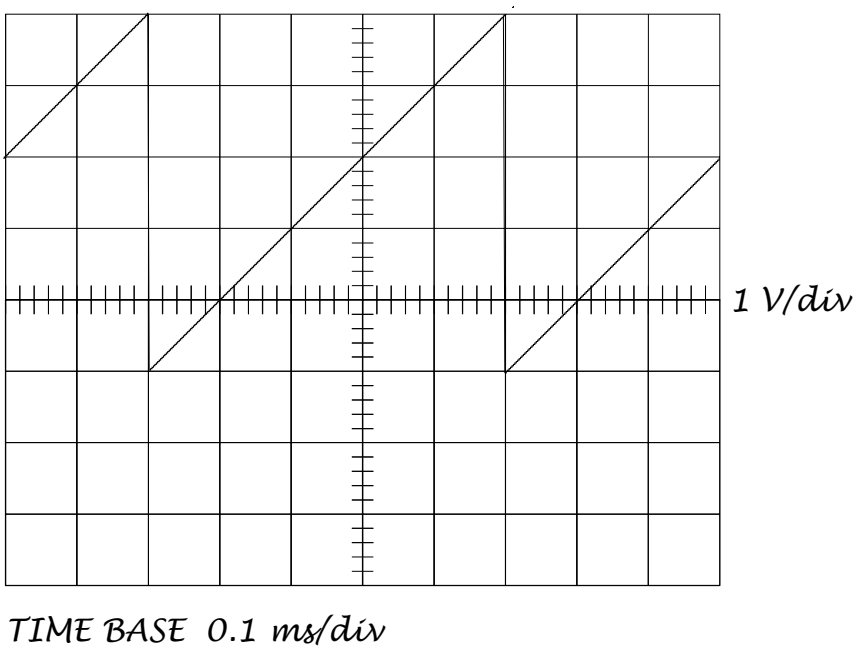


Fig. 14.13 Generating a continuous sawtooth using a MAX506 DAC.

loop cycle count of 6 cycles, gives a sawtooth duration of $(256 \times 6)/3 \approx 0.5$ ms at 2 μs per step.

Examples

Example 14.1

Augment the interrupt-driven ISR of Program 14.2 to implement a 16-deep buffer array of data to allow a limited mismatch between acquisition and reading rates.

Solution

One approach is shown in Fig. 14.14. A block of file registers is set aside by the programmer together with any other variables used by the programmer with a `cblock` directive of the form:

```
cblock
 ARRAY:16, OVERFLOW:1, BUF_EMPTY:1, ...  ; etc.
endc
```

and the File Select Register used as a buffer pointer to the next empty location in the array of samples.

In acquiring data the foreground ISR of Program 14.6 simply pushes the datum from the ADRES into the location pointed to by the FSR using

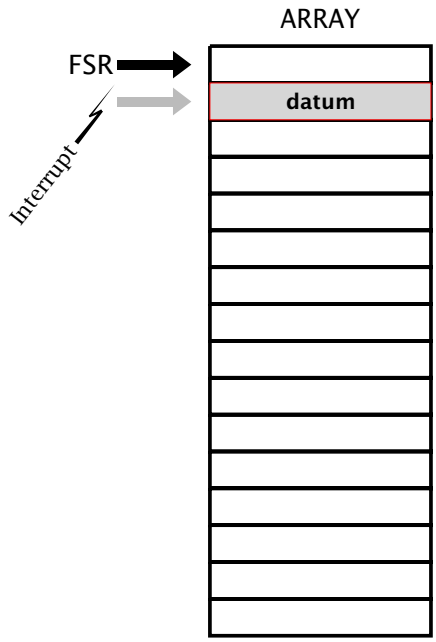


Fig. 14.14 Buffered data acquisition.

the indirect address mode and then increments this pointer ready for the next event.

The problem with this approach is that if the background program does not pull data out of the buffer quickly enough, decrementing the FSR, the buffer will overflow. As a consequence, if there are variables stored above ARRAY+15 then they will be overwritten. In order to avoid this problem, on overflow no more data should be saved and the state of the OVERFLOW file register set to non zero to show the background software that data has been lost.

Based on this ISR, a background routine to fetch data from the buffer would be something like this:

```
GET_IT  bcf    INTCON,GIE      ; Disable interrupts
        btfsc  INTCON,GIE      ; Make sure
         goto  GET_IT          ; IF not THEN DO again

        clrf   BUF_EMPTY       ; Zero Buffer-Empty flag
        movf   FSR,w           ; Check is FSR below ARRAY?
        sublw  ARRAY           ; ARRAY - FSR
        btfsc  STATUS,C        ; IF FSR is EQUAL or LESS THAN
         goto  CONTINUE        ; IF so THEN buffer is empty
        decf   BUF_EMPTY,f     ; ELSE show buffer is not empty
        movf   0,w             ; Get datum
        decf   FSR,f           ; Decrement buffer pointer
CONTINUE
        bsf    INTCON,GIE      ; and re-enable interrupt
```

Program 14.6 Buffered interrupt-driven data acquisition.

```
; ********************************************************
; * FUNCTION: ISR to read the A/D converter at EOC and put *
; * FUNCTION: in buffer if not full and update pointer     *
; * ENTRY   : On an interrupt.  FSR points to last entry   *
; * EXIT    : FSR incremented and new datum pushed into buf*
; * EXIT    : IF buffer is full OVERFLOW is returned as -1 *
; * EXIT    : ELSE returns zero                            *
; ********************************************************
; First save context
A_D_ISR movwf _work         ; Put away W
        swapf STATUS,w      ; and the Status register
        movwf _status

; ********************************************************

        btfss  PIR1,ADIF    ; Check; has there been a conversion
         goto  ISR_EXIT     ; IF not THEN false alarm

        incf   FSR,w        ; Move pointer up to the next location
        sublw   ARRAY+d'16'; FSR - (ARRAY+16) Outside the buffer?
        btfsc   STATUS,C    ; IF yes
         goto   FULL        ; THEN can't update

        clrf    OVERFLOW    ; Zero teh overflow flag
        incf    FSR,f       ; ELSE update pointer
        movf    ADRES,w     ; and get digitized byte
        movwf   INDF        ; and put in buffer
        goto    ISR_EXIT    ; and exit gracefully

FULL    movlw   -1          ; Show the world that buffer
        movwf   OVERFLOW    ; has overflowed

; ********************************************************

ISR_EXIT swapf  _status,w   ; Untwist the original Status reg
         movwf  STATUS
         swapf  _work,f     ; Get the original W reg back
         swapf  _work,w     ; leaving STATUS unchanged
         retfie             ; and return from interrupt
```

It is essential to avoid any alteration to the FSR during a background buffer fetch, so GIE is zeroed before and enabled after the process to disable interrupts. If the buffer pointer is at the bottom of the array then no update is carried out and the file register EMPTY is left at zero to show that the buffer was empty. Otherwise the datum pointed to is copied to the Working register; the buffer pointer is decremented and EMPTY is set to non zero.

Example 14.2

Using **C** coding show how a digitized reading from Channel 3 of a PIC16C74 can be acquired with the processor in its Sleep state.

Solution

The CCS compiler uses the `sleep()` function to put the MCU to sleep – this simply translates to the `sleep` instruction. A Sleep conversion cannot be implemented using the `read_adc()` function of Program 14.5 as this continuously polls the GO/$\overline{\text{DONE}}$ flag until it drops low. Instead we need to set and clear individual interrupt related bits before going to sleep in the manner outlined in the assembly-level Program 14.4. On wakening the state of ADRES can then be read 'manually'.

Program 14.7 Sleep conversion in **C**.

```
#include <16c74.h>
#bit ADIF = 0x0C.6   /* The A/D Interrupt Flag in PIR1[6]    */
#bit PEIE = 0x0B.6   /* The group interrupt flag in INTCON[6]*/
#bit GO  =  0x1F.2   /* The Go/NOT_DONE bit in ADCON0[2]     */

#define ADRES *(unsigned int *)0x1e
int main()
{
unsigned int i;
set_tris_a(0x0E);
setup_adc(ADC_CLOCK_INTERNAL);
setup_adc_ports(RA0_RA1_RA3_ANALOG);
set_adc_channel(3);

disable_interrupts(GLOBAL);/* Disable all ints (GIE & PEIE=1)*/
ADIF = 0;
enable_interrupts(INT_ADC);
PEIE = 1;            /* Enable the auxiliary group interrupts*/
/*    Code   */
GO = 1;
sleep();

i = ADRES;
}
```

Coding for this specification is shown in Program 14.7. Here the GO/$\overline{\text{DONE}}$, PEIE and ADIF bits are defined using the `#bit` directive. This time the script ADC_CLOCK_INTERNAL is used with the `setup_adc()` internal function to select the internal CR clock for the DAC module, as necessary for the Sleep conversion.

The internal function `disable_interrupts(GLOBAL)` clears *both* GIE and PEIE mask bits. The complementary `enable_interrupts(GLOBAL)`

sets both bits but we need to enable the PEIE only and leave GIE cleared. This is implemented by the 'bit-twiddling' statement `PEIE=1;`. Similarly, clearing the ADIF flag is directly actioned by `ADIF=0;`. Before calling `sleep()` the statement `GO=1;`manually starts the conversion. After `sleep()` the ADRES register is read giving the required digitized equivalent.

Example 14.3

The analog input channel voltage range for the PIC16C7X/71X devices[10] is limited to the positive range 0–V_{ref}, where V_{ref} can either be the internal V_{DD} voltage or an external voltage at RA3 in the range 3–V_{DD}. Many situations require a digitized mapping from bipolar analog signals. Design a simple resistive network to translate a bipolar voltage range of ±10 V to a unipolar range of 0–5 V, assuming V_{ref} is +5 V.

Solution

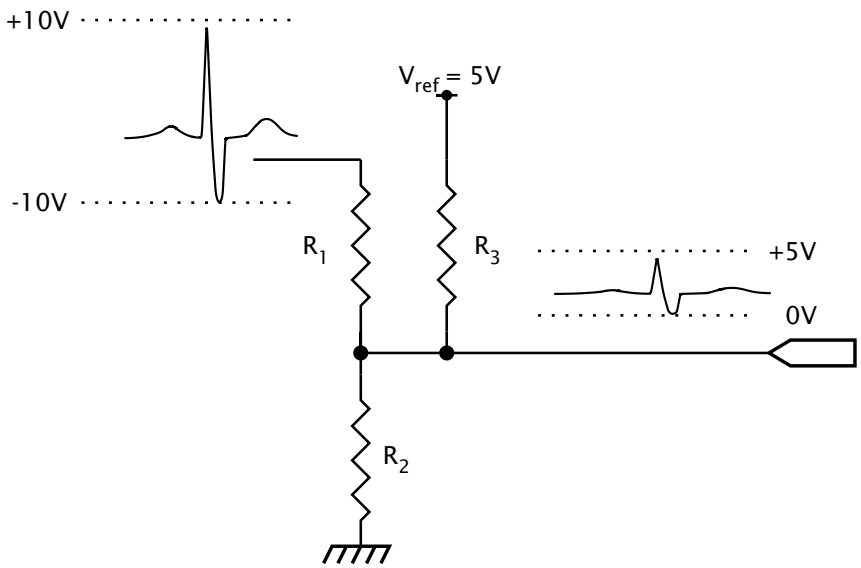


Fig. 14.15 A level-shifting resistor network.

One possibility is shown in Fig. 14.15. In calculating the value of the three resistors, the following transfer relationships must be adhered to:

1. The value of V_{ref} must appear at the summing node attenuated by 2 when V_{in} is zero; that is half scale. Thus a zero voltage 0 V $\mapsto$ 10000000*b*. To do this R_1 paralleled with R_2 must have the same resistance as R_3; i.e.:

$$R_3 = R_1 // R_2$$

[10] The 16C77X devices can be configured to accept bipolar input analog voltages.

2. V_{in} must be attenuated by the gain of the system G, where the input range is $\pm G \times V_{ref}$. In our case $G = 2$. Thus using the potential divider relationship:
$$2G = R_1 + (R_2 // R_3)/(R_2 // R_3)$$

After some manipulation we have:

$$\begin{aligned} R_1 &= (G-1) \times R_2 \\ R_2 &= G \times R_3 \end{aligned}$$

Of course we have three unknowns and only two equations so we have to start off by choosing a value for one of them below 10 kΩ, which is the maximum recommended value for input resistance. Picking 5 kΩ for R_3 gives R_2 as $2 \times 5 = 10$ kΩ and R_2 as $R_1/(G-1) = 10$ kΩ.

For the situation ± 5 V $\mapsto$ 5 V then $R_1 = \infty$ and $R_2 = R_3$.

Example 14.4

As part of a smart biomedical monitor, the peak analog value of an electro-cardiogram (ECG) signal is to be determined anew for each cycle. This R-point (see Fig. 7.1 on page 172) maximum value is to be output from Port B and RA5 is to be pulsed high whenever this value is being updated. Assuming that a PIC16C73/4 is used to implement the intelligence, the the ECG signal (conditioned as shown in Fig. 14.15) connected to Channel 1 RA1, devise a possible strategy. Timer 0 is being used to interrupt the processor at nominally 100 times per second (see Program 13.2 on page 370) design a suitable ISR to implement your strategy.

Solution

As in any biomedical parameter the ECG signal will vary from cycle to cycle in gain, shape and period. Even if this were not so, imperfections in the data acquisition system, notably the skin electrodes, can cause slow baseline (dc) drift. Thus the threshold at which the signal is to be tracked to its peak R-value must be reset at some sensible fraction of its previous peak during the period following the last update.

One possibility is shown in Fig. 14.16. Here the threshold is slowly decremented after the peak to ensure that a following peak of lower amplitude is not missed. On the basis of a lowest ECG rate of 40 beats per minute (period 1.5 s) if we reduce the threshold by one bit each two samples then the maximum reduction would be a count of 75 at a sample rate of 100 per second. To do this the threshold value THRESHOLD in Program 14.8 is stored as a double-byte number of form integer:fraction and half an integer (i.e. fraction = 10000000b) subtracted in each sample where the peak value MAXIMUM is not updated.

The task list implemented by this listing is:

1. DO a conversion to get ANALOG.

Program 14.8 ECG peak picking.

```
; ************************************************************
; * FUNCTION: ISR to update the ECG parameters               *
; * ENTRY   : On a Timer0 interrupt                          *
; * EXIT    : Update MAXIMUM and THRESHOLD:THRESHOLD+1       *
; ************************************************************
; First save context
ECG_ISR  movwf   _work          ; Put away W
         swapf   STATUS,w       ; and the Status register
         movwf   _status

; ************************************************************
         btfss   INTCON,T0IF    ; Was this a Timer0 interrupt?
          goto   ECG_EXIT       ; IF not THEN exit

         bcf  INTCON,T0IF    ; ELSE clear flag
         movlw   1              ; Initiate a conversion of
         call    GET_ANALOG     ; Channel 1

         movwf   TEMP           ; Save digitized value
         subwf   THRESHOLD,w    ; THRESHOLD - ANALOG
         btfsc   STATUS,C       ; IF no Borrow THEN
          goto   BELOW          ; don't update MAXIMUM
         movf    TEMP,w         ; ELSE get digitized value
         movwf   MAXIMUM        ; which is the new MAXIMUM
         movwf   PORTB          ; made visible to outside
         bsf     PORTA,5        ; which is signalled
         movwf   THRESHOLD      ; Now update double-byte
         clrf    THRESHOLD+1    ; threshold
         goto    ECG_EXIT       ; and finish

; Land here if the input is below the threshold
BELOW    bcf     PORTA,5        ; Signal no update
; Now reduce the threshold by 0.5 unless it is zero
         movf    THRESHOLD,f    ; Is integer threshold zero?
         btfsc   STATUS,Z       ; Skip if not
          goto   ECG_EXIT       ; IF it is THEN leave alone

         movlw   80h            ; 0.5 = 10000000b
         subwf   THRESHOLD+1,f  ; Take away from fraction byte
         btfss   STATUS,C       ; Skip if no borrow
          decf   THRESHOLD,f    ; ELSE decrement integer thres
; ************************************************************

ECG_EXIT swapf   _status,w    ; Untwist the original Status reg
         movwf   STATUS
         swapf   _work,f      ; Get the original W reg back
         swapf   _work,w      ; leaving STATUS unchanged
         retfie               ; and return from interrupt
```

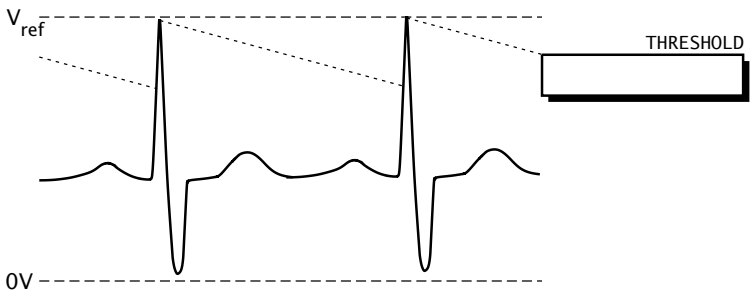


Fig. 14.16 ECG detection strategy.

2. IF (ANALOG > THRESHOLD)
   - MAXIMUM = ANALOG.
   - THRESHOLD = ANALOG.
   - PORTB = ANALOG.
   - RA5 = 1.
3. ELSE
   - Reduce THRESHOLD by 0.5.
   - RA5 = 0.

In updating THRESHOLD where ANALOG > THRESHOLD the integer byte takes the new value of MAXIMUM whilst the fractional byte is zeroed. Treating this byte pair as a 16-bit word, this effectively equates the threshold as MAXIMUM $\times$ 256 or THRESHOLD = MAXIMUM << 8, where MAXIMUM has been shifted left eight places. We are assuming that THRESHOLD has been zeroed in the background program during the initialization phase.

If the digitized analog sample is less than the threshold trip value then $80h = 10000000b$ is subtracted from the lower byte at THRESHOLD+1 and if this produces a borrow, then the upper byte at THRESHOLD is decremented. This subtract $\frac{1}{2}$ routine is skipped if the threshold has reached zero thus preventing underflow.

Although not shown, Program 14.8 uses the subroutine GET_ANALOG of Program 14.1 and its associated 12 μs delay subroutine.

Program 14.9 gives the **C** coded version implementing our task list. The `#int_rtcc` directive tells the compiler to treat the following function as a Real-Time Counter Clock (Timer 0) ISR. In function `ecg_isr()`, the variables `threshold` and `maximum` are declared `static`. This means that their value will be retained after the function has exited and will be available next time on entry. The default way of treating **C** function variables is to hold their value only for the duration of the function. An alternative way of dealing with this problem is to declare such variables outside any function, in which case they will be global and retain their value indefinitely.

`threshold` is defined as a `long int` and the CCS compiler will then treat this datum as a 16-bit variable as required. The definition in equating `threshold` to zero will only initialize it once when the program begins

Program 14.9 An implementation of the ECG peak picker in **C**.

```
#int_rtcc
ecg_isr()
{
unsigned int analog;
static unsigned long int threshold = 0;
static unsigned int maximum;
analog = read_adc();

if(analog > threshold>>8)
    {
    maximum = analog;           /* New maximum value          */
    PORT_B  = analog;           /* Show the outside world     */
    threshold = maximum << 8;   /* New 2-byte threshold       */
    output_bit(PIN_A5,1);       /* Tell outside world         */
    }
else
    {
    threshold = threshold - 0x0080; /* Reduce by 0.5          */
    output_bit(PIN_A5,0);       /* Signal no update           */
    }
}
```

its run, as the variable is `static`. Again this is not the normal behavior of the default `auto` variable.

In equating `threshold` to the new `maximum` value, the latter is multiplied by 256 by shifting left eight times – a good compiler will automatically change a `N*256` to `N<<8`. This double-byte form allows for the reduction by half a bit $0080h$ to give the specified falling trip level.

Both implementations assume that the analog module, ports and interrupt mask bits have been set up at the beginning of the background program as described earlier in the chapter.

Self-assessment questions

14.1 In Example 14.4 the decay of the threshold level was linear. Although this is fairly effective for situations where the nominal period is known a priori and does not vary greatly, an exponential decay would be better suited where this is not the case. To generate this type of relationship a fixed *percentage* of the value at each sample point should be subtracted to give the new outcome rather than a constant. Show how you could modify Programs 14.8 and 14.9 to decrement at a rate of approximately 0.4% ($\approx \frac{1}{256}$) on each sample and determine the time constant in terms of the number of samples.

14.2 A programmer writing an ISR-based handler for an analog module has replaced the `retfie` instruction in Program 14.2 by `return`. What effect will this have?

14.3 Real world analog signals are noisy. In practice this means that some form of filtering or smoothing is frequently required. In any circumstance noise coming in from outside should not have any appreciable frequency components above half the sampling rate since such noise will be frequency shifted back into the baseband as shown in Fig. 14.4. Such low-pass filtering must be applied to the signal *before* the A/D conversion.

Although this external **anti-alias** filter must by definition be implemented using hardware circuitry (such as a CR network), noise within the passband can be smoothed out using software filtering routines. One simple approach to digital filtering is to take multiple readings and average them to give a composite outcome. For example, 16 readings summed and shifted right four times ($\div 16$) would reduce random noise by a factor of $\sqrt{16} = 4$.

Another approach well known to staticians, is to take a moving average; for example, of a stock price over a month interval. An efficient algorithm of this type is a 3-point average:

$$\texttt{Array[i]} = \frac{S_n}{4} + \frac{S_{n-1}}{2} + \frac{S_{n-2}}{4}$$

where S_n is the nth sample from the analog module.

Show how you could modify the GET_ANALOG subroutine to remember the last samples from the two previous calls and return the smoothed value.

14.4 It has been suggested that as part of the ECG monitor of Example 14.4 that a MAX506 DAC be used to introduce an automatic gain control function preceding the PIC's analog input. The aim of the AVC is to keep the peak of the analog input between $\frac{3}{4}$ and $\frac{7}{8}$ full scale. How might you go about implementing this subsystem? Hint: Remember that each channel of a MAX506 is the product of its digital input and V_{ref} and that the latter can vary between 0 V and V_{DD}.

14.5 How could the time between ECG peaks be measured with a resolution of 10 ms and output at one of the parallel ports as an extension of Example 14.4?

14.6 An input analog sinusoid signal, conditioned as shown in Fig. 14.15, is to be full-wave rectified; that is voltages that were originally negative are to have their sign changed. Design a routine to do this assuming that the input voltage is available at ADRES and the processed output is to be presented via Port B to a DAC.

14.7 Write a **C** coded program after Program 14.5 to compare two analog voltages at Channel 0 and 1 bringing RA2 high when $V_0 > V_2$, RA3 likewise for $V_0 == V_1$ and RA4 for $V_0 < V_1$. Assume a PIC16C71 device with a clock frequency of 10 MHz. The PIC16C71 can be set to make RA0 and RA1 analog (RA0_RA1_ANALOG) with the rest digital.

14.8 Figure 14.17 is based on Fig. 10 of Microchip's application note AN546 *Using the Analog-to-Digital (A/D) Converter* as a means of providing an external voltage reference source for power-sensitive applications. How do you think the circuit works and what factors govern the choice of current limiting resistor?

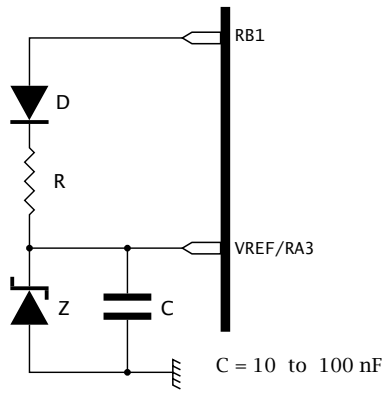


Fig. 14.17 A controllable external voltage circuit.

14.9 Microchip recommend that where possible channle 0 RA0/AN0 should not be used in the PIC16C71 due to possible noise problems. Can you see why this is so and can you think of any way around this problem?

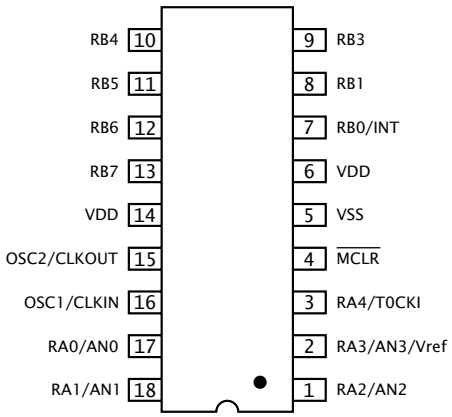


Fig. 14.18 Pinning for the PIC16C71.

CHAPTER 15

To Have and to Hold

Several mid- and high-range PIC devices feature a small EEPROM scratchpad memory that can be controlled and accessed indirectly via Special-Purpose Registers (SPRs) in the same manner as other peripheral devices. An integral non-volatile scratchpad enables the programmer to read and modify static data, such as the odometer tally in a car, which needs to be retained in the absence of a power supply – see Example 12.3 on page 351. Although this facility can be implemented using an external EEPROM memory, such as the 24LC01 of Fig. 12.22 on page 352, where only a modest amount of non-volatile data needs to be stored, integral EEPROM storage increases reliability and reduces cost, size and power requirements.

Our objective here is to examine the non-volatile storage facilities available to members of the mid-range PIC family. After reading this chapter you will:

- Be familiar with the characteristics of the EEPROM Data memory.
- Know how to both read and write data to the EEPROM module.
- Understand how the main flash EEPROM Program memory can be used in some devices to store and retrieve non-volatile data.
- Be able to contrast the EEPROM Data module and flash Program memory as a location for non-volatile data.

The PIC16C83/4 introduced in 1994, was the first PIC device to use EEPROM technology for its main Program store. As we gave seen in Fig. 2.12 on page 27, Electrically Erasable PROM is similar to EPROM but does not require UV radiation to erase data. Although EEPROM technology is more expensive than EPROM, its use in implementing the Program store is convenient in prototyping and educational/hobbyist applications. Along with this innovation, an EEPROM peripheral module was featured which enabled the programmer to store up to 64 bytes of non-volatile data independently of the normal file register memory.

The PIC16C83/4 and its analogous flash EEPROM memory successor, the PIC16F83/4, remained the only EEPROM family member until the introduction of the PIC16F87X in 1998. As of 2000, Microchip were committed to introduce flash EEPROM versions of most of their standard mid- and high-range devices. Thus, for example, the PIC16F74 will shadow the PIC16C74 processor.

In this chapter we will use the 18-pin PIC16F83/4, 28-pin PIC16F873/6 and 40-pin PIC16F874/7, here denoted the PIC16C8X and PIC16F87X lines respectively, as our exemplar. However, before examining the details, it is instructive to look at an application requiring the use of non-volatile storage. A good example of this is the smart card of Fig. 12.1 on page 305. Here we need to store, amongst others, the card account number, PIN number, start and expiry dates. Some of this data, such as the account number, is essentially fixed. Security data may need to be altered occasionally by the user from a terminal. If the card is used as a cash card its credit will need to be charged via an ATM and discharged when payments are made. The size and cost sensitivities of a smart card processor is such that *integral* EEPROM data storage is highly advantageous.

Figure 15.1 shows the logic organization of the PIC16F8X EEPROM Data module.[1] The memory matrix is not part of the normal Data and Program stores but is indirectly accessed via four SPRs which address the target byte, collect/hold data and control the read and write processes.

EEPROM matrix
The mid-range EEPROM Data module architecture supports 256 byte cells. The bottom 64 locations are implemented in both the PIC16F83 and PIC16F84 devices. The PIC16F873/4 has a capacity of 128 bytes and the PIC16F876/7 implement all 256 memory cells. Key features are:

- 1,000,000 minimum (10^7 typical) Erase/Write cycle endurance for each cell.
- Maximum Erase/Write cycle time 8 ms.
- Data retention greater than 40 years.

EEPROM ADdRess register EEADR
The EEADR register located at File $09h$ can address up to a maximum of 256 bytes of EEPROM data. Where less than maximum capacity is implemented, unused upper address bits must be 0 to ensure that the address is within the physical address space. In the PIC16F8X allowable addresses are in the range $00–3Fh$.

EEPROM DATA register EEDATA
The EEDATA register located at File $08h$ either holds the 8-bit datum read out of the addressed cell or the byte the programmer wishes to write to the target EEPROM cell.

EEPROM CONtrol register 1 EECON1
The EEPROM module has two modes of operation, with EECON1 located at File $88h$ in Bank 1 controlling and monitoring the Read and Write cycles – see Fig. 15.2.

[1] The earlier PIC16C83/4 had an identical architecture.

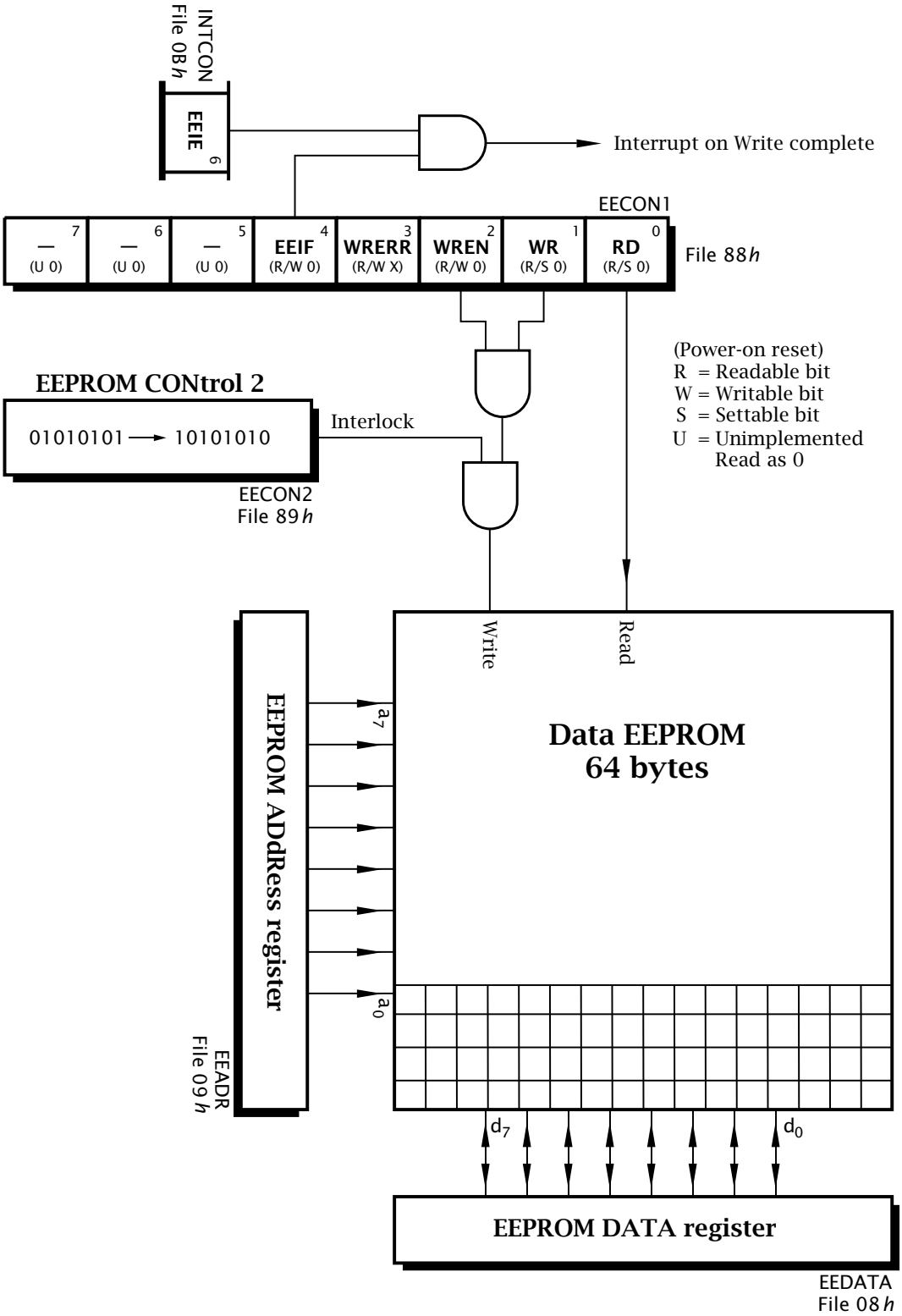


Fig. 15.1 The PIC16F8X Data EEPROM module.

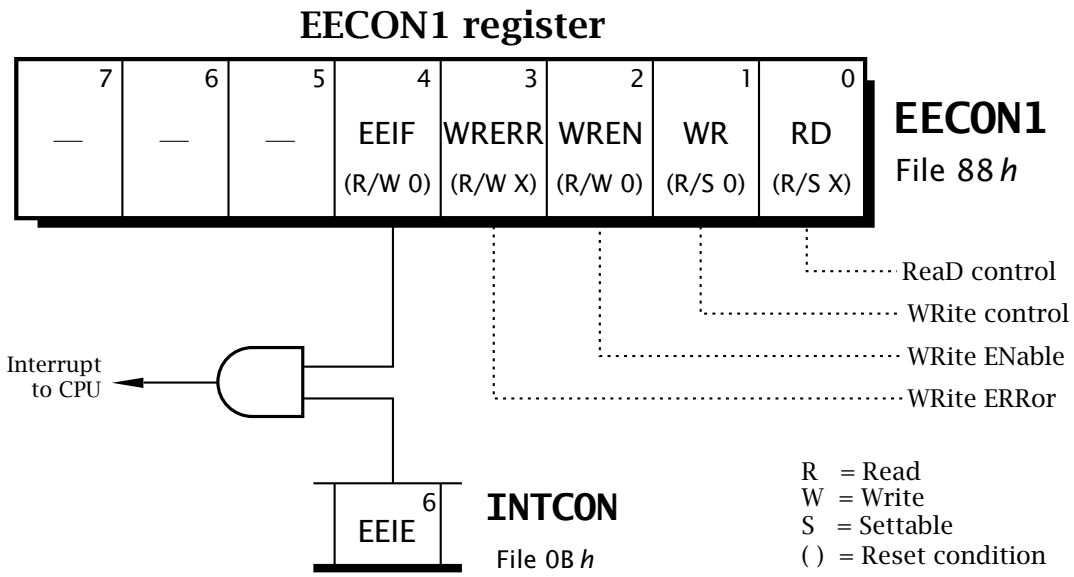


Fig. 15.2 The PIC16F8X EECON1 register.

EEPROM CONtrol register 2 EECON2
This register located at File 89*h* is not physically implemented – it always reads as zero. Rather the action of writing the successive code pattern 01010101 → 10101010 *with no interruption* is used to unlock the Write cycle. This arcane incantation is deliberately designed to convolute the process as security against unintended alterations in the data.

In order to read a specified datum from the EEPROM module we have to implement software to execute the task list:

1. Copy the target cell's address to EEADR.
2. Set RD to 1 to initiate the Read cycle.
3. RD is automatically cleared immediately and the target 8-bit datum can be read from EEDATA any time from the next instruction cycle as convenient.

Program 15.1 Retrieving a byte from the EEPROM Data module.

```
; ************************************************************
; * FUNCTION: Gets one byte from the EEPROM Data module      *
; * ENTRY   : Address in EEADR                               *
; * EXIT    : Datum in W and in EEDATA. System in Bank0      *
; ************************************************************
EE_GET  bsf    STATUS,RP0  ; Change to Bank1
        movlw  b'00000001' ; Set RD for Read cycle
        movwf  EECON1      ; Read datum into EEDATA
        bcf    STATUS,RP0  ; Back to Bank0
        movf   EEDATA,w    ; Copy into W
        return             ; for return
```

Subroutine EE_GET in Program 15.1 directly implements this process and illustrates the return of the datum from the EEPROM cell to the Working register. The datum will remain in EEDATA until the register is reused.

Writing data to the EEPROM Data module is deliberately made more Byzantine to reduce the chance of a spurious Write corrupting the data due to a software bug or processor malfunction because of, say, a power glitch. The task list to write a datum to a specified cell is:

1. Copy the target cell address to EEADR.
2. Set WREN in EECON1[2] to enable the Write process.
3. Disable all interrupts.
4. Send 55$h$to EECON2.
5. Send AA$h$to EECON2.
6. Set WR to initiate the Write cycle.
7. Clear WREN.
8. Enable interrupts.
9. Wait until WR returns to zero, signalling the completion of the Write cycle, and exit.

The Write cycle will not initiate if the interlock sequence items 4 – 6 is not exactly followed without interference. For example; in an interrupt-driven system an interruption during the interlock sequence will abort the Write cycle. Thus in this situation interrupts should be disabled by clearing GIE until the Write cycle has been initiated, otherwise this step can be omitted.

If desired the completion of the Write cycle can be used to interrupt the processor. This is enabled by setting the EEIE mask bit in INTCON[6]. When the interrupt flag EEIF, located in EECON1[4], is set in the normal way then the interrupt is generated. It should be cleared in the ISR.

It is possible that the processor is reset, for example by a Watchdog overflow, before the Write cycle is complete. In this situation, the EEPROM datum may be corrupt. The **WRERR** flag in EECON1[3] will be set if the Write operation has been prematurely terminated with a Reset action. If this is not the case, when the cycle is complete the datum may be read back and verified to give extra security. The WREN bit may be cleared at this point to help prevent an accidental Write. Doing this before the Write is complete will not affect the operation.

Program 15.2 implements this task list. Both EEDATA and EEADR are set up by the caller program with the byte data and address. The subroutine is not pulled out until the Write cycle has completed; typically 4 ms. This ensures that these SPRs will not be altered during the cycle which may possibly give an erroneous outcome.

In order to illustrate these concepts we will repeat Example 12.3 on page 351 replacing the external serial EEPROM with the internal module. We will assume that the odometer count is located at EEPROM cells 10 – 12h.

Program 15.2 Putting a byte into the EEPROM Data module.

```
; ************************************************************
; * FUNCTION: Writes one byte into the EEPROM Data module    *
; * ENTRY   : Datum byte in EEDATA, module address in EEADR  *
; * EXIT    : Interrupts disabled for 9 instructions         *
; * EXIT    : System in Bank0                                *
; ************************************************************
EE_PUT  bsf    STATUS,RP0  ; Change to Bank1
        bsf    EECON1,WREN ; Enable for Write cycle
EE_LOOP bcf    INTCON,GIE  ; Disable all interrupts
        btfsc  INTCON,GIE  ; Check, did it clear?
         goto  EE_LOOP     ; IF not THEN do again
        movlw  55h         ; Now do the interlock
        movwf  EECON2
        movlw  0AAh
        movwf  EECON2
        bsf    EECON1,WR   ; Initiate the Write cycle
        bcf    EECON1,WREN ; Optionally disable any other Writes
        bsf    INTCON,GIE  ; Re-enable interrupts
EE_EXIT btfsc  EECON1,WR   ; Check, has the Write completed?
         goto  EE_EXIT     ; IF not THEN retry
        bcf    STATUS,RP0  ; Go back to Bank0
        return             ; and return when cycle has finished
```

The coding shown in Program 15.3 makes use of the two subroutines EE_GET and EE_PUT to read and subsequently write the three odometer bytes from the EEPROM Data module. The address of the first (highest) byte is copied into EEADR at the beginning of the subroutine and is subsequently incremented and decremented in situ to point to the appropriate datum.

Once the 3-byte odometer state has been fetched and copied into memory it is incremented in exactly the same manner as in Program 12.15 on page 354. The augmented array is then written back into EEPROM in the opposite sense as it was read, with EEADR being decremented. The EE_PUT subroutine checks that the Write cycle has been completed before returning and thus timing need not be checked by the calling program.

As well as altering data under program control it is possible to initialize the state of the EEPROM Data module when the executable program is being externally blasted into the Program memory; as illustrated in Fig. 10.5(a) on page 261. The area of Program memory beyond the user Program store belongs to the special test/configuration memory space $2000h$– $30FFh$ and can be accessed only during external programming. In Fig. 10.5(b) we observed that the Configuration fuse word is located at $2007h$. The EEPROM module also lies in this space located at $2100h$–$21FFh$. For example, to store the value of sine every 10° between 0° and 90° as part of the program source code we have:

```
     org 2100h     ; The EEPROM Data module
SINE de  0, 2Ch, 57h, 7Fh, 0A4h, 0C4h, 0DDh, 0F0h, 0FBh, 0FFh
```

where the assembler directive **de** (Data EEPROM) specifies the comma delimited list of data. Once the PIC has been programmed, the contents of the EEPROM Data module will look like Fig. 15.3.

Program 15.3 Incrementing the non-volatile odometer count in Data EEPROM.

```
; ************************************************************
; FUNCTION: Adds one onto the triple-precision odometer total*
; RESOURCE: Subroutines EE_GET and EE_PUT                    *
; ENTRY   : Current total in EEPROM module at 10:11:12h      *
; EXIT    : Incremented total back in EEPROM module          *
; EXIT    : also available in RAM at LSB:NSB:MSB             *
; ************************************************************
EXTRA_MILE
         movlw  10h       ; Address of high-byte odometer total
         movwf  EEADR     ; Copy into EEPROM address register
         call   EE_GET    ; Read byte from EEPROM module
         movwf  MSB       ; and put into file register MSB
         incf   EEADR,f   ; Address of middle byte odometer
         call   EE_GET    ; Read byte from EEPROM module
         movwf  NSB       ; and put into file register NSB
         incf   EEADR,f   ; Address of low byte odometer
         call   EE_GET    ; Read byte from EEPROM module
         movwf  LSB       ; and put into file register LSB

; Now increment 3-byte array
         incf    LSB,f    ; Add one
         btfss   STATUS,Z ; Is it now zero
          goto   PUT_BACK ; IF not THEN continue
         incfsz  NSB,f    ; Increment middle byte
          goto   PUT_BACK ; IF not zero THEN continue
         incf    MSB,f

; Put the augmented odometer count back in Data EEPROM
PUT_BACK movf    LSB,w    ; Get new odometer low byte
         movwf   EEDATA   ; Put in EE Data register

         call    EE_PUT   ; Write to EEPROM cell 12h
         decf    EEADR,f  ; Address of middle byte
         movf    NSB,w    ; Get new odometer middle byte
         movwf   EEDATA   ; Put in EE Data register
         call    EE_PUT   ; Write to EEPROM cell 11h
         decf    EEADR,f  ; Address of high byte
         movf    MSB,w    ; Get new odometer low byte
         movwf   EEDATA   ; Put in Data register
         call    EE_PUT   ; Write to EEPROM cell 10h
         return
```

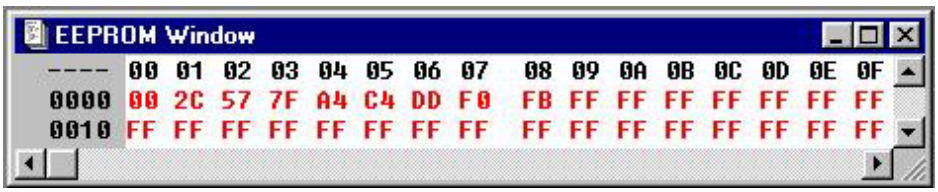


Fig. 15.3 The first 32 bytes of EEPROM holding the sine look-up table.

Any data programmed in the way can be subsequently read by the program. For example, to read sin(50) the contents of EEPROM module location $05h$ is read, giving from our diagram $C4h$ or 196 decimal ($\frac{196}{256} = 0.76525$).

Although such load-time data is non-volatile it can be altered from within the program at run time. The Code Protection fuse does not secure data in the EEPROM Data module from modification, only code in the true Program store is protected.

Although it is possible to initialize the Program store in a similar manner using the `dw` (Data Word) directive in a similar manner, as shown in Program 15.6, this is of little use as the Harvard architecture's separation of Data and Program store memory spaces means that there is no way an instruction can access this data.[2] However, newer 16-bit PIC devices with a Flash EEPROM Program store allow the program to read and write such data in a similar indirect manner to that used for the EEPROM Data module.

To illustrate these devices we will be using the PIC16F87X line. Like the PIC16F8X PICs, these devices all have a main Program store implemented using Flash EEPROM technology as well as a standard EEPROM Data module. Besides the EEPROM module they have the same range of integral peripheral devices described in previous chapters, but slightly enhanced. For example, the Analog module has a 10-bit resolution and the Synchronous Serial port has hardware Master I^2C modes.

To cope with the extra SPRs and Data memory the Data store is organized into four banks. Those relevant to EEPROM facilities are located in Banks 2 and 3 and so both `RP0` and `RP1` bits in the Status register need to used to change bank, as described on page 108.

Four devices are represented in this line:

PIC16F873
This 28-pin device has a 4 Kbyte flash Program store and 128-byte EEPROM Data store together with a 192-byte file register store. It is pin compatible with the PIC16C73 device.

[2] The PIC17CXXX and 18CXXX 16-bit core families have `tablrd` and `tablewt` instructions to read and write respectively data from/to the Program store.

PIC16F874
This 40-pin device has a 8 Kbyte flash Program store and 256-byte EEPROM Data store together with 192 file registers. It is pin compatible with the PIC16C74 device.
PIC16F876
This 28-pin device is the same as the PIC16F873 but with twice the Program store capacity at 8 Kbytes and double the EEPROM Data module capacity at 256 bytes The file register store is increased to 368 file registers.
PIC16F877
This is a 40-pin version of the PIC16F876.

Key EEPROM properties are:

- 100,000 minimum EEPROM Data module Erase/Write cycle endurance per cell.
- 1000 minimum flash EEPROM Program store Erase/Write cycle endurance.
- Maximum Write/Erase time 8 ms (typical 4 ms) for both the Data module and flash memory.

Of particular note is the endurance limit of 1000 Write cycles for the flash EEPROM. Whilst this is entirely satisfactory when changing the device's program, it is a limitation for some non-volatile data storage situations. Thus flash Program memory storage is more applicable to constant data, such as the sine lookup table, rather than for information that requires frequent update, such as the odometer.

Flash EEPROM has a smaller geometry than normal EEPROM. Whilst this speeds up its operation, charges which eventually trap in the floating gate insulation have a disproportional effect on the storage mechanism and leads to earlier deterioration.

Figure 15.4 shows the PIC16F87X EEPROM Data module with the flash Program store superimposed. This form of representation is used as the EEDATA and EEADR registers are common to both EEPROM arrays. Of course, the flash Program store is larger both in the number of cells (8 Kbytes against 256 bytes) and in cell size (14 bits against 8 bits). Thus both Data and Address registers have the high-end extensions **EEDATH** and **EEADRH** respectively to cope with this additional capacity.

As we shall see, the process of reading from and writing to either array is similar. The target module is chosen using the **EEPGD** (EEProgram/$\overline{\text{Data}}$) control bit control bit in EECON1[7]. Apart from this additional bit and the removal of the EEIF interrupt flag to the PIR2 register, the EECON1 register of Fig. 15.5 is identical to the basic PIC16F8X version shown in Fig. 15.2. The virtual EECON2 interlock register remains the same.

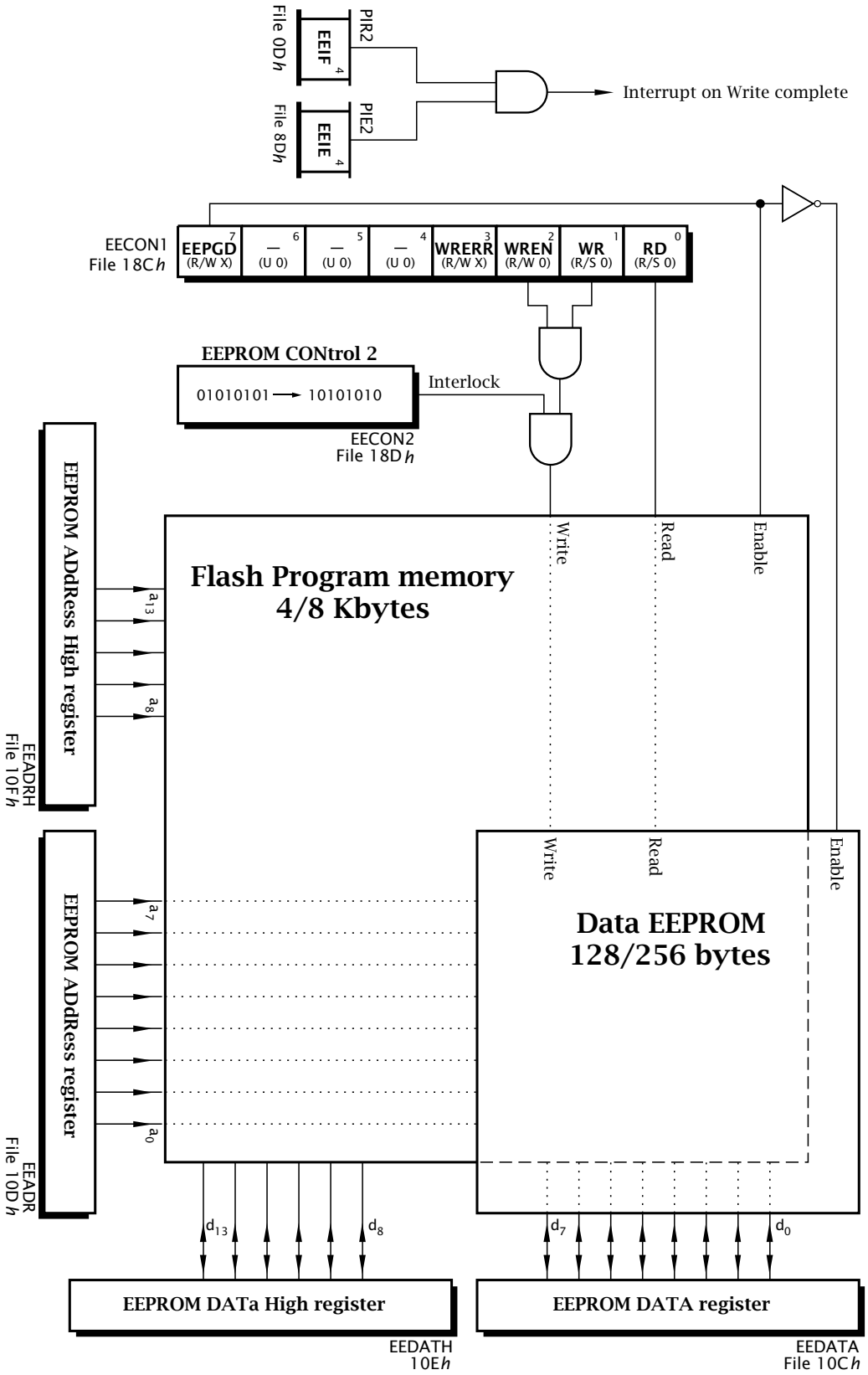


Fig. 15.4 The PIC16F87X flash and Data EEPROM storage system.

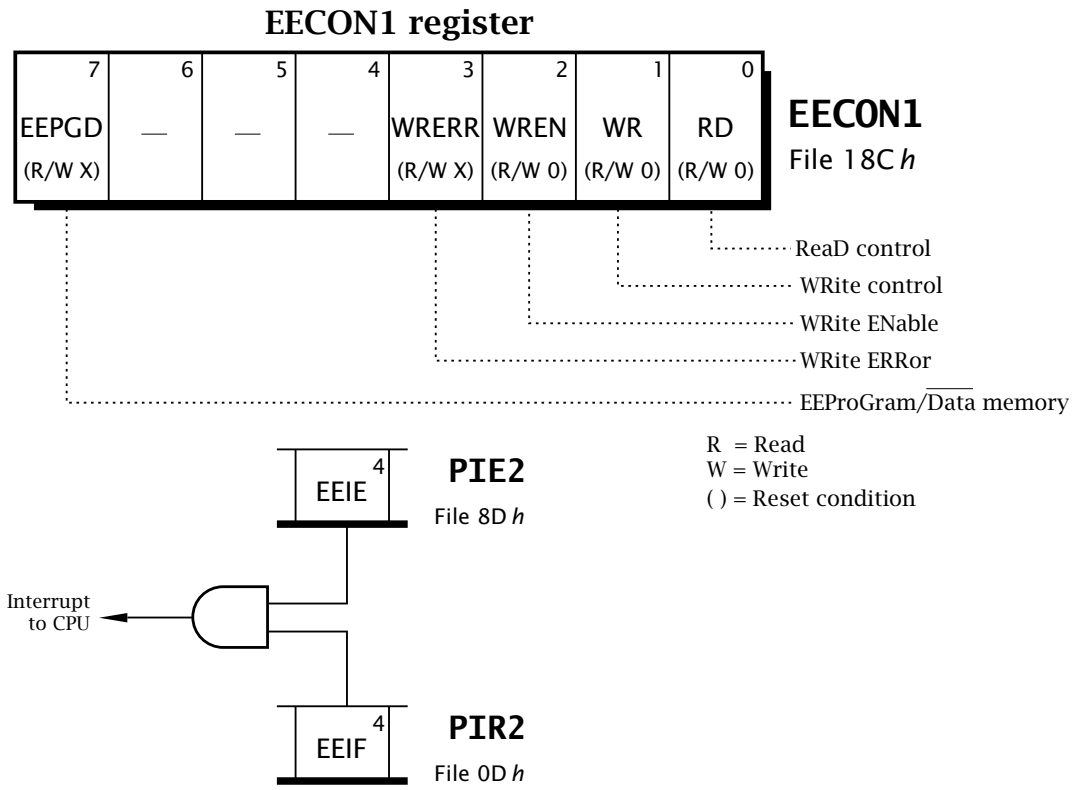


Fig. 15.5 The PIC16F87X EEPROM Control register 1.

Reading and writing to the EEPROM Data module is identical to that used for the more basic PIC16F8X device. The only change necessary to subroutines EE_GET and EE_PUT relates to the Bank 2 location of EEDATA and EEADR and Bank 3 for EECON1 and EECON2.

The process of reading from the flash Program store is similar to that of the EEPROM Data module but using double Data and Address registers. However, we are interacting with the same Program store from which code is being fetched into the execution unit. In consideration of this dualism, two dummy nop instructions should follow the instruction setting the RD bit in EECON1[0]. This gives our Read task list for the flash Program store.

1. Copy the target cell's address to EEADRH:EEADR on entry.
2. Set RD to 1 to initiate the Read cycle.
3. Execute two dummy nop instructions.
4. RD is immediately cleared automatically and the 14-bit target datum can be read from EEDATH:EEDATA as convenient.

The subroutine FLASH_GET of Program 15.4 implements this process assuming that the cell address has been placed in EEADRH:EEADR on entry.

Program 15.4 Reading a word from the flash Program store.

```
; ************************************************************
; * FUNCTION: Gets one byte from the flash Program store     *
; * ENTRY   : Address in EEADRH:EEADR                        *
; * EXIT    : Datum in EEDATH:EEDATA. System in Bank0        *
; ************************************************************
FLASH_GET
       bsf   STATUS,RP1    ; Change to Bank3
       bsf   STATUS,RP0
       movlw b'10000000'   ; Point to Program memory
       movwf EECON1        ; by setting EEPGD
       bsf   EECON1,RD     ; Set RD for Read cycle
       nop                 ; Dummy nops
       nop
       bcf   STATUS,RP1    ; Return to Bank0
       bcf   STATUS,RP0
       return
```

The Write cycle also is virtually identical to its Data module counterpart but with the addition of a double-nop relaxation phase. This gives us our flash Write cycle task list:

1. Copy the target cell address to EEADRH:EEADR.
2. Set WREN in EECON1[2] to enable the Write process.
3. Disable all interrupts.
4. Send 55$h$to EECON2.
5. Send AA$h$to EECON2.
6. Set WR to initiate the Write cycle.
7. Execute two dummy nop instructions.
8. Clear WREN.
9. Enable interrupts.
10. Wait until WR returns to zero, signalling the completion of the Write cycle, and exit.

The subroutine FLASH_PUT in Program 15.5 assumes that the cell address is in EEADRH:EEADR and 14-bit datum is in EEDATH:EEDATA on entry.

For our example we will design a subroutine that will return the square of an integer between 0 and 100 in EEDATH:EEDATA. We could of course calculate this by multiplication, but for the purposes of this exercise we will implement this exercise as a look-up table located in flash Program store. As this is a table of constants we can load the data into flash memory at the same time as the rest of the program code.

In Program 15.6 the table is located at 300h in the Program store. The directive **dw** (Data Word) is similar to de but each datum in the comma separated list is 14-bits. For convenience the **radix** directive is used to specify constants by default are treated as decimal.

Program 15.5 Writing to flash Program memory.

```
; ********************************************************
; * FUNCTION: Writes one byte into the flash Program store   *
; * ENTRY   : Datum byte in EEDATH:EEDATA                    *
; * ENTRY   : Datum address in EEADRH:EEADR                  *
; * EXIT    : Interrupts disabled for 11 instructions        *
; * EXIT    : System in Bank0                                *
; ********************************************************
FLASH_PUT  bsf   STATUS,RP0  ; Go to Bank 3
           bsf   STATUS,RP1
           bsf   EECON1,EEPGD; Target the flash Program store
           bsf   EECON1,WREN ; Enable for Write cycle
FLASH_LOOP bcf   INTCON,GIE  ; Disable all interrupts

           btfsc INTCON,GIE  ; Check, did it clear?
            goto FLASH_LOOP  ; IF not THEN do again
           movlw 55h         ; Now do the interlock
           movwf EECON2
           movlw 0AAh
           movwf EECON2
           bsf   EECON1,WR   ; Initiate the Write cycle
           nop               ; Dummy nops
           nop
           bcf   EECON1,WREN ; Disable any more Writes
           bsf   STATUS,GIE  ; Re-enable interrupts
FLASH_EXIT
           btfsc EECON1,WR   ; Check, has the Write completed?
            goto FLASH_EXIT  ; IF not THEN retry
           bcf   STATUS,RP1  ; Go back to Bank 0
           bcf   STATUS,RP0
           return            ; & return when cycle has finished
```

Directly following the table is the executable code. In this manner Program 15.6 is comparable to a **C++** `class` where a program object comprises both data members and member functions (subroutines).

The subroutine itself builds up the table element `nn` address by placing the integer passed in `W` in `EEADR` and the constant $03h$ in `EEADRH`. This gives the double-byte address as $3nnh$. Once this is done, the subroutine `FLASH_GET` retrieves the 14-bit datum from the table. The subroutine then moves both bytes from `EEDATH`:`EEDATA` and returns the datum in the two file registers `SQRH`:`SQRL` in Bank 0. Unlike the PIC16F8X, general-purpose file registers are not reflected across the various banks, so each byte copied from EEPROM SPRs in Bank 2 needs switching to Bank 0 once the byte has reached the Working register.

When the program has been burnt into flash memory by the external programmer the Program store in the area around $300h$ will look like Fig. 15.6.

Program 15.6 Squaring an integer.

```
      __config _CPD_OFF & _WRT_ENABLE_OFF
      org      300h
; ************************************************************
; * FUNCTION: Generates the square of an integer             *
; * RESOURCE: Subroutine FLASH_GET                           *
; * ENTRY   : Integer in W range 0 -- 100                    *
; * EXIT    : 14-bit square in SQRH:SQRL. In Bank0           *
; ************************************************************

TABLE
 dw 0,1,4,9,16,25,36,49,64,81,100,121,144,169,196,225
 dw 256,289,324,361,400,441,484,529,576,625,696,729,784,841
 dw 900,961,1024,1089,1156,1225,1296,1369,1444,1521,1600,1681
 dw 1764,1849,1936,2025,2116,2209,2304,2401,2500,2601,2704
 dw 2809,2916,3025,3136,3249,3364,3481,3600,3721,3844,3969
 dw 4049,4225,4356,4489,4624,4761,4900,5041,5184,5329,5476
 dw 5625,5776,5929,6084,6241,6400,6561,6724,6889,7056,7225
 dw 7396,7569,7744,7921,8100,8281,8464,8649,8836,9025,9216
 dw 9409,9604,9801,10000

SQUARE bsf   STATUS,RP1    ; Move to Bank2
       bcf   STATUS,RP0
       movwf EEADR         ; Build up the address
       movlw 3
       movwf EEADRH
       call  FLASH_GET     ; Get table entry n in 3nnh
       bsf   STATUS,RP1    ; Move back to Bank2
       bcf   STATUS,RP0
       movf  EEDATA,w      ; Get lower byte of square
       bcf   STATUS,RP1    ; Bank0
       movwf SQRL          ; Copy to SQRL in Bank0
       bsf   STATUS,RP1    ; Back to Bank2
       movf  EEDATH,w      ; Get high byte of square
       bcf   STATUS,RP1    ; Bank0
       movwf SQRH          ; and copy to SQRH in Bank0
       return
```

Like the PIC16F8X, as discussed on page 260, the PIC16F87X line has code protection fuses in its configuration word – as shown in Fig. 15.7. The primary function of code protection is to prevent the external programmer reading code from the Program store to give a measure of security against unauthorized peeking at the code. In the case of the PIC16F87X devices two code bits (duplicated as bits 13:12 and 5:4) in the configuration word in the special/test configuration area at 2007*h* give protection for all the Program store (00), the top half of the store (01), the top 256 bytes only (10) or no protection (11); the default situation. If protection is given to *any* area of memory then the external programmer cannot subsequently write data into anywhere in the Program store nor

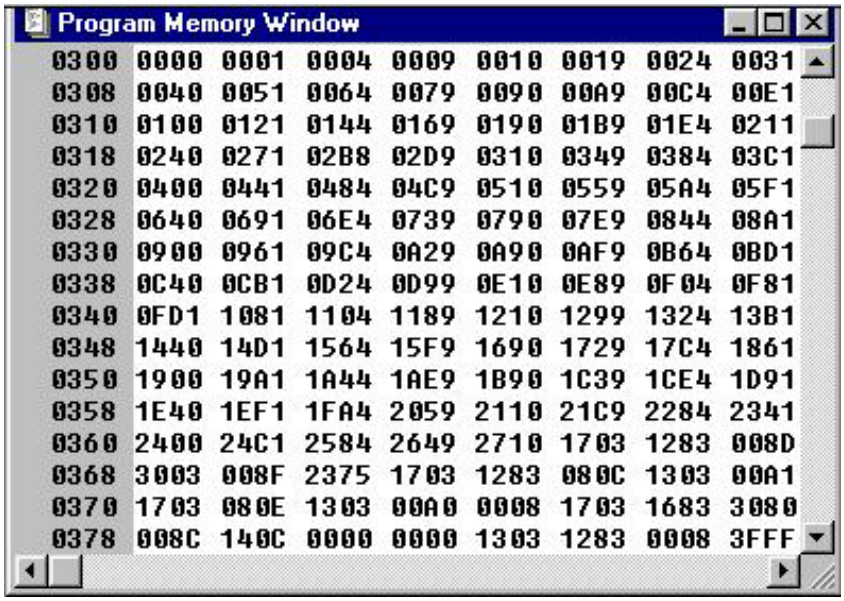


Fig. 15.6 View of the flash Program module showing the look-up table and subroutine SQUARE.

erase it. Thus protection should be disabled during prototyping! However, reading is only inhibited in protected areas.

Code protection also affects internal Writes into the Program store using code such as in Program 15.5. Internal Writes can be made into unprotected areas of Program memory provided that the WRT fuse is 1; its default setting. Setting this fuse to 0 (_WRT_ENABLE_OFF) will disable internal Writes irrespective of the main code protection setting. Internal Reads are not affected by code protection. Program 15.6 shows the __config directive used to disable all code protection in Program memory; which is actually superfluous as this is the default situation.

Unlike the PIC16F8X, the PIC16F87X devices can also protect the EEPROM Data module from internal Writes. The CPD fuse (Code Protection Data module) if set to 0 will inhibit any changes in this data.

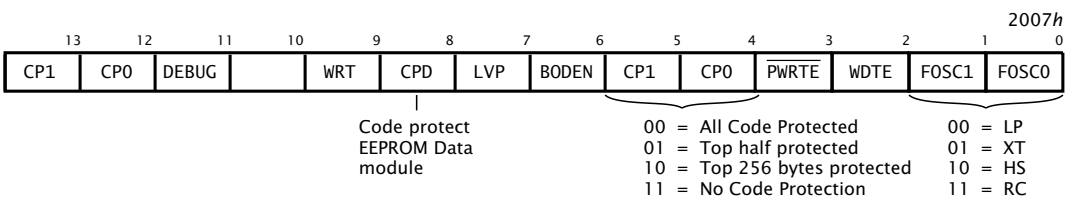


Fig. 15.7 Configuration word for the PIC16F87X devices.

Although the table has here been placed at an even 256-byte boundary for convenience, in practice it can be located anywhere in memory. In the general case the table offset nn will need to be added to the 14-bit address TABLE. SAQ 15.2 discusses how this address arithmetic could be done.

Examples

Example 15.1

The CCS compiler has the following built-in functions dealing with the EEPROM Data module:

read_eeprom(address)
Reads a byte from the specified EEPROM address.

write_eeprom(address, value)
Write the value to the specified address and returns only when the Write cycle has finished.

Write a **C** function to duplicate the odometer update implemented at assembler level in Program 15.3.

Solution

Like its assembly-level counterpart of Program 15.3, the function of Program 15.7 is divided into three phases.

1. This phase creates an array of three bytes named odometer[] which will act as a temporary store for the odometer count located in EEPROM. As the EEPROM Data module handles data in discrete byte

Program 15.7 C-based coding for the odometer.

```
void odometer(void)
{
unsigned int odometer[3];        /* Define the 3-byte count  */
odometer[0] = read_eeprom(0x10); /* Get the existing count   */
odometer[1] = read_eeprom(0x11);
odometer[2] = read_eeprom(0x12);

/* Increment array                                           */
if(++odometer[0] != 0) break;
else if(++odometer[1] != 0) break;
else odometer[2]++;

/* Now return incremented count to the EEPROM                */
write_eeprom(0x10, odometer[0]);
write_eeprom(0x11, odometer[1]);
write_eeprom(0x12, odometer[2]);
}
```

packages, it is best to model the object in **C** in the same manner. The array is given the current reading by extracting the data one byte at a time from EEPROM locations 10:11:12*h* using the `read_eeprom()` function.

2. Once the 3-byte data is in the Data store it is incremented using an `if-else` tree:
   (a) Increment low byte and check for zero. IF not zero THEN the overall addition is complete ELSE pass on carry to next byte.
   (b) Increment the middle byte and check for zero. IF not zero THEN the overall addition is complete ELSE pass on carry to next byte.
   (c) Increment most significant byte.
3. Finally each byte is written back into the EEPROM Data module using the `write_eeprom()` function.

Comparing the hand-coded assembly of Program 15.3 against the code generated from Program 15.7 gives 52 instructions against 91.

Example 15.2

A feasibility study is being undertaken in using a PIC-based Sauna controller. This is to monitor temperature and control heating and cooling units. There is also to be an over-temperature emergency alarm and shut-off.

One possibility is to use an 8-pin PIC with integral A/D conversion, such as the PIC12C672, together with an external temperature transducer. Someone has suggested that an efficient approach to this problem would be to use the variation in the internal Watchdog timer's period with temperature as a cost effective, albeit crude, sensor.

Experimental data was collected using a sample of eight production devices from the same manufacturing lot, with a soak time of 30 minutes at each tested temperature and 500 uncalibrated periods averaged to produce the graph of Fig. 15.8.

The data presented in Fig. 15.8 is based on Microchip's application note AN720 *Measuring Temperature Using the Watch Dog Timer (WDT)*. The two loci give the maximum and minimum across the range of tested devices. The Watchdog period was measured in Timer 0 overflows counting an internal 4 MHz clock. The Watchdog timer used a 1:8 prescale ratio.

From this data it can be seen that there is a correlation between period and rising temperature. However, although the overall trend is predictable, different devices will have varying offsets and slopes. For example, within the eight devices tested here the scale factor (scalar) varies from 2.28–2.42 counts per degree Celsius. This will necessitate a calibration phase before the system is used. If the count at one temperature T_0 and the scalar are known for any device, then for the specific case where a count $COUNT_n - COUNT_0$ is recorded:

$$\Delta T = (COUNT_n - COUNT_0) \times \text{Scalar}$$

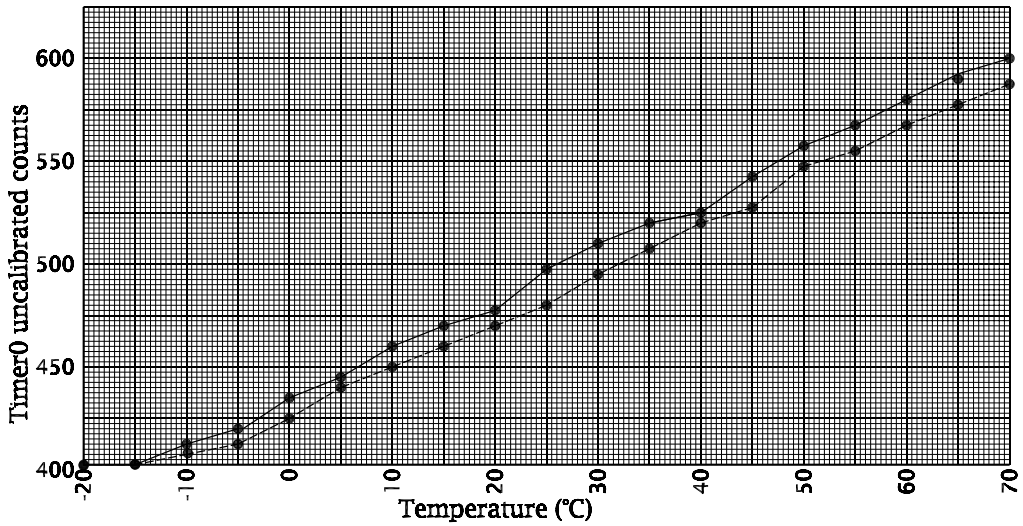


Fig. 15.8 Watchdog timer period versus temperature.

In order to calibrate these devices it has been decided to soak batches in a refrigerator at 0°C and save the 2-byte count in the EEPROM data module of a PIC16F83. The soak test is to be repeated in an oven at 30°C with the *difference* between this second count and the original value to be stored in a single EEPROM byte. Once this has been done, the device can be reprogrammed with the final running program overwriting the original calibrate program. This running program can then calculate temperature using the actual Watchdog period COUNT_n:

$$T = (\mathrm{COUNT}_0 - \mathrm{COUNT}_n) \times (\mathrm{Difference}/30)$$

where COUNT_0 is the 2-byte EEPROM datum showing the count at 0°C and Difference is the 1-byte EEPROM datum showing the change in count over 30°C.

Show how you could code the calibrate program to implement this specification.

Solution

Five tasks can be identified which must be undertaken at the end of a Watchdog period.

- Average the current Timer 0 roll-over count with the existing low-temperature (i.e. 0°C) count.
- Average the current Timer 0 roll-over count with the existing high-temperature (i.e. 30°C) count.
- Store the low-temperature count in EEPROM.
- Calculate the difference between low- and high-temperature count and store in EEPROM.
- Do nothing.

Port lines can be used to signal which of the first four active actions are to be actioned. For example; if RA0 is high then add the current Timer 0 roll-over count to the existing 2-byte low-temperature count. Unless this is the first reading, divide by two to give an average. The complete batch of devices could have their RA0 pin held high for a few minutes 30 minutes after the refrigerator stabilizes at 0°C.

Bringing RA0 low and then RA2 high for a short time signals an EEPROM storage action. With all port lines low, no action is taken representing both time before temperature stabilization and after EEPROM programming.

The Timer 0 and Watchdog timers are initialized together with the count values on a once-only basis on Power-up reset. This will typically only occur when the devices are powered up when in the temperature bath. Subsequent resets will normally be due to Watchdog time-outs. The state of the Status register's $\overline{\text{TO}}$ flag can be used to ascertain the source of reset – see page 363.

Program 15.8 shows the routine used to initialize the timers and variables entered if $\overline{\text{TO}}$ is 1 on reset; that is on Power-up. Once this is done, the system enters an endless loop `goto $` (the assembler replaces the label `$` by the instruction's address) which simply keeps going to itself!

Also shown is the ISR servicing a Timer 0 interrupt. This increments the double-byte variable ROLL_OVER:ROLL_OVER+1 and this is the count value read by the system on a Watchdog reset giving a numerical value for period.

Eventually the Watchdog timer will time-out and reset the processor. This time $\overline{\text{TO}}$ will be 0 and the routine labelled READING in Program 15.9 will be entered. This checks the state of each of the four RA3:0 pins in turn, executing one of the four listed tasks. If no pin is high, the program simply clears ROLL_OVER:ROLL_OVER+1 and Timer 0, the Watchdog timer is restarted and an endless loop entered. This READING_EXIT routine is also entered at the end of the four tasks.

The first two tasks are shown in Program 15.9. Here the 2-byte Timer 0 roll-over count is either added to the existing value LO_TEMP:LO_TEMP+1 or HI_TEMP:HI_TEMP+1 as appropriate and the outcome shifted once right to divide by two to give the average. As the count total is modest, 2-byte arithmetic is sufficient to avoid overflow. If this is repeated over a duration of several minutes an averaged value will result.

If this is the very first time a reading has been made then the divide by two operation is skipped and the flag variable FIRST_LO or FIRST_HI as appropriate is made non zero.

The core routine with respect to this chapter is given in Program 15.10. If RA2 is 1 the 2-byte low-temperature count LO_TEMP:LO_TEMP+1 is copied into the bottom two bytes of the EEPROM Data module using the EE_PUT subroutine of Program 15.2.

Program 15.8 The Sauna Power-up reset sequence and ISR.

```
        __config   _WDT_ON & _CP_OFF & _RC_OSC

        cblock 20h
         _work:1, _status:1
         FIRST_HI:1, FIRST_LO:1
         ROLL_OVER:2, LO_TEMP:2, HI_TEMP:2
         DELTA_TEMP:1
        endc

        org    0
START   goto   MAIN
        org    4
        goto   ISR

MAIN    btfss  STATUS,NOT_TO  ; IF Watchdog timeout
         goto  READING        ; THEN must have a reading

        clrwdt
        movlw  b'11011010'    ; Wdt enabled with a 1:8 prescale
        bsf    STATUS,RP0     ; Change to Bank1
        movwf  OPTION_REG     ; and TMR0 internal clock
        bcf    STATUS,RP0     ; and back to Bank0
        clrf   FIRST_HI
        clrf   FIRST_LO
        bsf    INTCON,T0IE    ; Enable Timer0 interrupt
        clrf   TMR0           ; Zero the Timer
        clrf   ROLL_OVER+1    ; Zero the 2-byte Timer roll-over
        clrf   ROLL_OVER
        bsf    INTCON,GIE     ; Enable all interrupts
        goto   $              ; Endless loop

; ************************************************************
; * The ISR to increment the 2-byte COUNT IF TMR0 interrupt    *
; ************************************************************
; First save context in usual way
ISR       movwf    _work        ; Put away W
          swapf    STATUS,w     ; and the Status register
          movwf    _status

; ************************************************************
; The core code
          incf     ROLL_OVER+1,f; Record one more roll-over
          btfsc    STATUS,C     ; Skip if no carry
           incf    ROLL_OVER,f  ; Increment upper byte
          bcf      INTCON,T0IF  ; Clear interrupt flag
; ************************************************************
         swapf    _status,w    ; Untwist the original Status reg
         movwf    STATUS
         swapf    _work,f      ; Get the original W reg back
         swapf    _work,w      ; leaving STATUS unchanged
         retfie                ; and return from interrupt
```

If RA3 is 1 the difference between the 2-byte high and low temperatures is then calculated. With the data shown in Fig. 15.8 it can be seen that a 30°C difference will not exceed a byte's worth of storage so only the

Program 15.9 Reading a new period count.

```
READING btfsc   PORTA,0       ; Check; new low temp desired?
         goto   NEW_LO        ; IF yes THEN go to it!
        btfsc   PORTA,1       ; Check; new high temp desired?
         goto   NEW_HI        ; IF yes THEN go to it!
        btfsc   PORTA,2       ; Check; update low temp desired?
         goto   UPDATE_LO     ; IF yes THEN go to it!
        btfsc   PORTA,3       ; Check; update high temp desired?
         goto   UPDATE_HI     ; IF yes THEN go to it!
        goto    READING_EXIT  ; ELSE nothing doing

NEW_LO  movf    ROLL_OVER+1,w; ELSE get low byte TMR0 roll-over
        addwf   LO_TEMP+1,f   ; and add it to low byte low temp
        btfsc   STATUS,C      ; Check for Carry
         incf   LO_TEMP,f     ; IF so THEN record it
        movf    ROLL_OVER,w   ; Now get high byte of roll-over
        addwf   LO_TEMP,f     ; and add it to high byte low temp
        movf    FIRST_LO,f    ; Is this the 1st low reading?
        btfsc   STATUS,Z
         goto   FIRST_TIME_LO; IF so THEN go to it!
        rrf     LO_TEMP,f     ; ELSE divide sum by two
        rrf     LO_TEMP+1,f
        goto    READING_EXIT  ; and finished
FIRST_TIME_LO                 ; IF first reading simply transfer
        incf    FIRST_LO,f    ; No longer the first reading
        goto    READING_EXIT

NEW_HI  movf    ROLL_OVER+1,w; ELSE get low byte TMR0 roll-over
        addwf   HI_TEMP+1,f   ; and add it to low byte high temp
        btfsc   STATUS,C      ; Check for Carry
         incf   HI_TEMP,f     ; IF so THEN record it
        movf    ROLL_OVER,w   ; Now get high byte of roll-over
        addwf   HI_TEMP,f     ; and add it to high byte high temp
        movf    FIRST_HI,f    ; Is this the 1st high reading?
        btfsc   STATUS,Z
         goto   FIRST_TIME_HI; IF so THEN go to it!
        rrf     HI_TEMP,f     ; ELSE Divide sum by two
        rrf     HI_TEMP+1,f
        goto    READING_EXIT  ; and finished
FIRST_TIME_HI                 ; IF first reading simply transfer
        incf    FIRST_HI,f    ; No longer the first reading

READING_EXIT
        clrf   TMR0           ; Zero the timer
        clrwdt                ; Reset the Watchdog timer
        clrf   ROLL_OVER+1    ; Zero the roll-over count
        clrf   ROLL_OVER
        goto   $              ; Wait for another Watchdog reset
```

Program 15.10 Updating the Sauna EEPROM.

```
UPDATE_LO
       movf     LO_TEMP,w    ; Get high byte of low temperature
       bsf      STATUS,RP0   ; To Bank1
       movwf    EEDATA       ; In EEPROM Data register
       clrf     EEADR        ; EEPROM address 00h
       call     EE_PUT       ; Write datum in
       movf     LO_TEMP+1,w  ; Get low byte of low temperature
       bsf      STATUS,RP0   ; To Bank1
       movwf    EEDATA       ; Is new datum
       incf     EEADR,f      ; EEPROM address 01h
       call     EE_PUT       ; Write Datum in
       goto     READING_EXIT

UPDATE_HI ; Work out HI_TEMP-LO_TEMP & store at 02 in EEPROM
; Only need to subtract the lower bytes as diff fits in one byte
       movf     HI_TEMP+1,w   ; Get low byte high temperature
       subwf    LO_TEMP+1,w   ; Subtract low byte low temperature
       movwf    DELTA_TEMP    ; giving the difference

       bsf      STATUS,RP0    ; To Bank1
       movwf    EEDATA        ; Delta temperature in 02h
       movlw    2
       movwf    EEADR
       call     EE_PUT
       goto     READING_EXIT
```

lower byte of the subtraction is implemented. This single byte difference is then written into EEPROM in the normal way.

Self-assessment questions

15.1 Good program practice dictates that the datum written into Data EEPROM should be verified as the value that was intended to be written. Show how you could modify the EE_PUT subroutine of Program 15.2 to return a value -1 in a file register if the action is not successful, otherwise zero.

15.2 In Program 15.6 we placed the look-up table at a 256-byte boundary in the Program store (specifically 300h) to simplify the computation of the 1-byte table index. Thus to look up table entry nn we simply place the address 3nn$h$ in the EEADRH:EEADR register pair.

Placing program segments at user-defined addresses is never a good idea, as subsequential program alterations can cause code to overlap unless care is taken. Letting the assembler sort out loca-

tions of labels is much more reliable. However, in our case we would need to add nn to the address the assembler selects for the label TABLE. Unfortunately Program store addresses are 13-bits wide and PIC arithmetic is only 8-bit. Microchip compatible assemblers have the directives high and low to separate the upper and lower bytes parts of a label; eg. movlw low TABLE. Using these directives modify the subroutine SQUARE if the directive org 300h is removed.

15.3 Microchip-compatible assemblers have the directive da (DAta) which can be used to store strings of character codes in Program memory. For example:

```
MESSAGE  da "Hello world/n",0
```

which places the codes characters in quotes coded in 7-bit ASCII code packed two at a time in each 14-bit word followed by all zeros. The \n escape character means New Line – ASCII code 0Ah.

Assuming that this is done in a PIC16F87X device, write a subroutine called PDATA (Print DATA) to fetch each character from Program memory and transmit to a terminal using the subroutine PUTCHAR of Program 12.11 on page 340.

15.4 A certain hotel security system is to use a PIC-based reprogrammable smart card for electronic guest room locks. On registration the card is to be charged up with the following details:

1. A 4-digit room number, eg. 1311.
2. Start data, eg. 13072000.
3. End date, eg. 15072000.

Assume that the PIC has an integral EEPROM Data module and communicates with the receptionist's terminal via a serial input subroutine, such as described in Program 12.11 on page 340. Data is coded in ASCII in the order outlined, preceded with the character STX, terminated by ETX and delimited by SP – see Table 1.1 on page 5. Design a routine to interpret the data and store them in EEPROM.

CHAPTER 16

A Case Study

Up to this point our microcontroller material has been presented piecemeal. To complete our study we are going to put much of what we have learnt to good use and design both the hardware and software of an actual widget (gadget). This is not an easy task to do in a single short chapter. However, very little new material needs to be presented at this point, rather a process of coalescence.

We begin with our specification. Students invariably talk too long during their oral presentations. It is proposed that a dedicated embedded microcontroller-based system be designed to act as a time monitor. This monitor should default to a time-out of 10 minutes, but will have the provision to vary the allotted time from 1 to 99 minutes.

Once triggered, the monitor should perform the following sequence of operations:

1. When the GO switch is closed, a green lamp will illuminate and a dual seven-segment display will show a count-down from the time-out value to 03 at one-minute intervals.
2. After a further minute, an amber lamp only will illuminate, the count of 02 will be displayed and a buzzer will sound for nominally one second.
3. After a further minute, a red lamp only will illuminate together with a display of 01. The buzzer will sound for two seconds.
4. Finally, after another minute the display will show 00, the red lamp will continue to be illuminated and the buzzer will sound continuously until the STOP switch is pressed. This will halt the timer and turn off all displays, lamps and buzzer. Indeed, closing the STOP switch at any time during the sequence above will cause the system to permanently halt. The system may be restarted from the time-out value by resetting the processor.
5. At any time the sequence can be frozen by toggling the PAUSE switch. When toggled again, the sequence will continue on from where it left off.
6. In order to alter the time-out from the default value of 10, the SET switch must be closed when the system is reset. The display will then show 99 and will count down slowly. The value showing when the

SET switch is released will be the new time-out and will be retained indefinitely until another set process.

The first decision to be made is the choice of microcontroller (MCU). In this case we are constrained by the need to use our book's model device, i.e. one of the mid-range PIC family. As we require non-volatile storage to store the time-out value we are limited at the time of writing to the PIC16F8X and 16F87X lines of devices to avoid the necessity to use an external serial EEPROM. As the display will physically be large for long-distant viewing the bulk of the circuit is not critical so the cheaper PIC16F84 device is our choice. The 18-pin count of this device compared to the alternative 40-pin PIC16F874 will require additional support functions to expand the port pin budget, but should better illustrate the trade-offs of more complex systems.

Based on this decision the final target hardware is shown in Fig. 16.1. The port pin budget is allocated as follows:

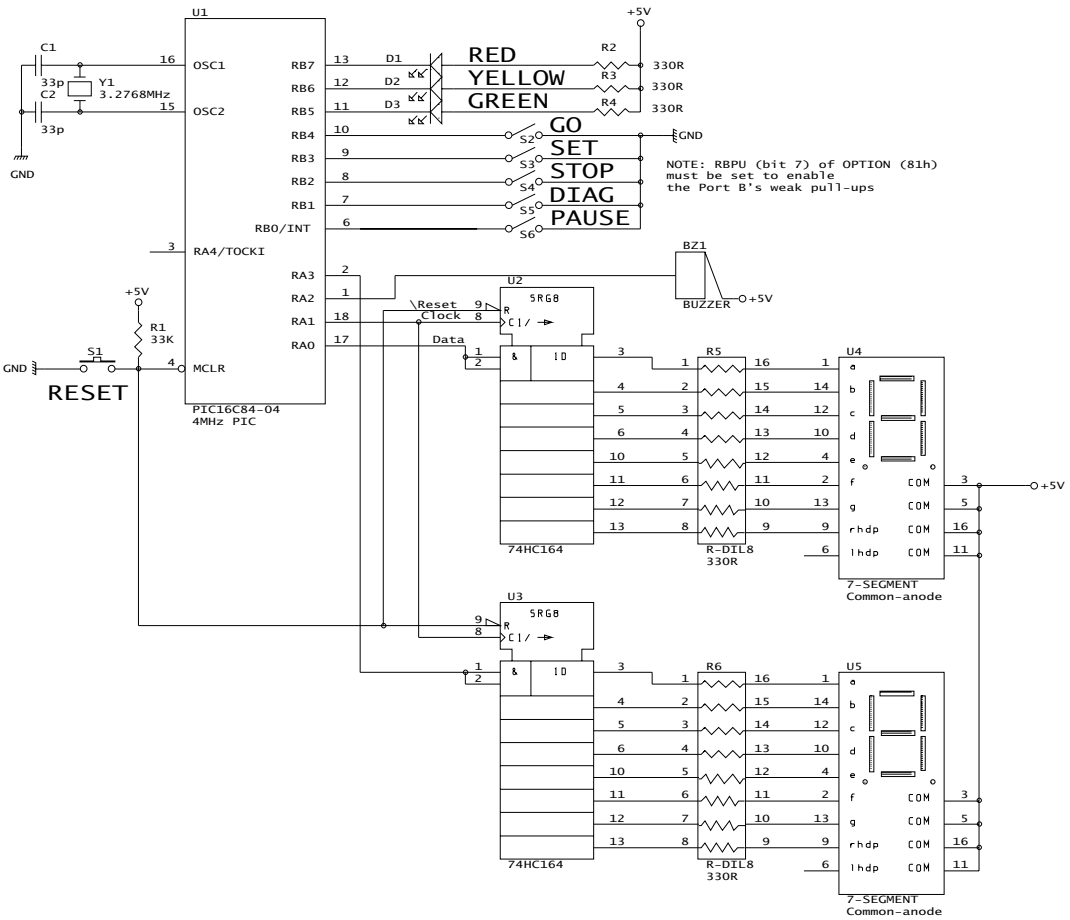


Fig. 16.1 The annunciator hardware.

Switches
The five switches S2...S6 implementing the functions GO, SET, STOP, DIAG, PAUSE are read from Port B at RB[4:0]. By using this port's internal pull-up resistors (see Fig 11.7 on page 280) no external resistors are required.

S1 with R1 provides a Manual reset in order to restart the count. This $\overline{\text{MCLR}}$ signal also provides a Reset signal feed for external circuitry.

All six switches can be conveniently implemented as momentary contact keyboard switches.

Lamps
Three suitably colored 10 mm (0.4") high-brightness LEDs D3...D1 driven from RB[7:5] provide the light signals. 330Ω series resistors limit the current to nominally 10 mA.

Buzzer
The buzzer should be a miniature solid-state device. A typical piezo-electric implementation will operate over a wide d.c. voltage range of typically 3–16 V and require little more than 1 mA at 5 V.[1]

The buzzer is driven via RA2.

Numerical display
Two 7-segment displays give the required 2-digit read-out, facilitating the maximum specified period of 99 minutes. As only four port pins remain, a serial interface is implemented. This is similar to that shown in Fig. 12.2 on page 307 but each SIPO shift register has a separate data feed, with RA0 being used for the ten's digit and RA3 for the units digit. Both digits can therefore be simultaneously updated with eight shifts.

The common-anode seven-segment display pinning shown in the diagram is that of the 16-pin Dual In Line (DIL) footprint with both left and right decimal points – lhdp and rhdp. Only the latter is used here (to indicate that the system has paused) in conjunction with the 8-bit 74HC164 shift register. Alternative 16- and 14-pinouts are commonly available and even dual-digit packages. However, even the 16-pin footprint pinout is not standardized.

Smaller-sized displays, typically below 0.8″/20 mm, use a single LED for each bar, with a conducting voltage drop of around 2 V.[2] The DIL 330 Ω series resistors R5 and R6 limit the current to around 10 mA. The common anodes are connected directly back to the normal +5 V power supply to avoid current surges affecting the logic circuits, and should be decoupled by small tantalum capacitors. Although the displays are normally rated for 20 mA, restricting the current to this value gives suf-

[1] If you want to put paid to any possibility of the speaker continuing, a piezo-electric sound bomb producing 110 dB at 1 m distance needs a 12 V d.c. supply at 200 mA.

[2] Larger displays, e.g. 2.24″/56 mm, have typically two or four LEDs in series. In the latter case a separate 12 V supply would be needed and current buffering.

ficient illumination and means that the 74HC164 shift registers do not need current buffering.[3]

Crystal
A 3.2768 MHz crystal provides the timing for the MCU's clock oscillator, giving an instruction rate of 819.21 kHz. A typical crystal of this value has a tolerance of ±30 ppm and temperature coefficient of ±50 ppm.

This unusual choice is $2^{16} \times 50$ so if we use the 8-bit Timer 0 with a prescale value of 1:64 then we can create an interrupt 50 times per second. An alternative low-power configuration would be to use a 32.768 kHz crystal and generate an interrupt every two seconds. However, compared to the current consumption of the optical components, the MCU's power dissipation is minor.

With the hardware environment designed, we can now concentrate on the software.

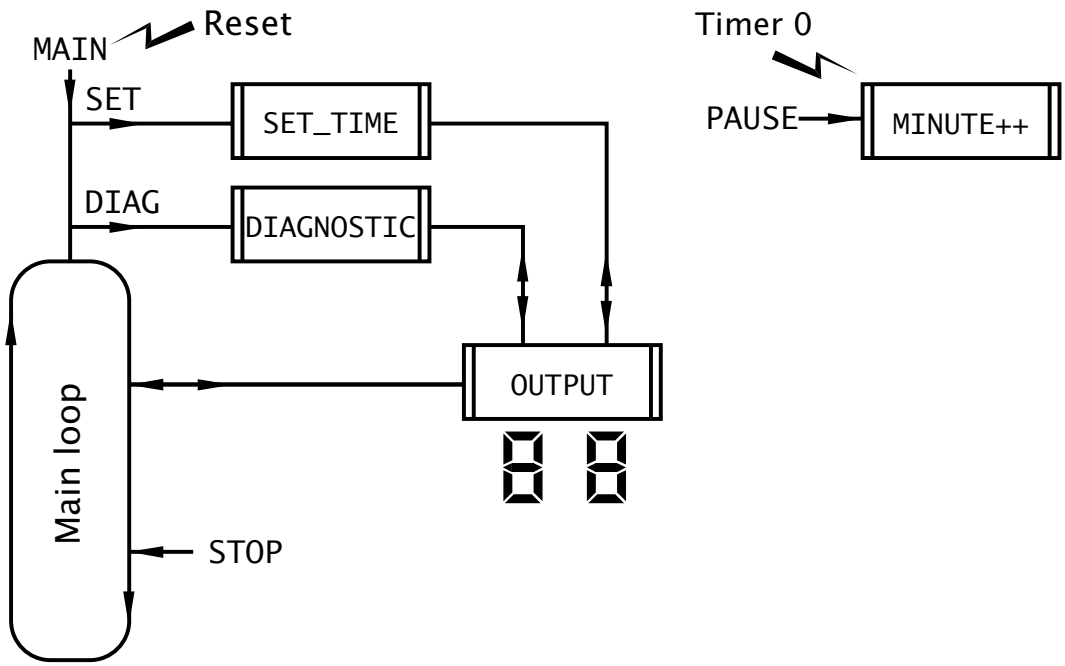


Fig. 16.2 The modular software structure.

Figure 16.2 shows the basic modular structure for our system. Here the distinctive double right/left edged box denotes a subroutine or Interrupt Service Routine (ISR). Three distinct processes can be identified together with two major supporting tasks.

[3] Alternatively low-current seven-segment displays are available.

Timebase task
All processes are time related. Timekeeping is implemented in hardware by generating an interrupt 50 times each second. By keeping a Jiffy count, seconds and minute tasks are updated and are used to sequence the appropriate process.

By monitoring the PAUSE switch this decrementing time chain can be by-passed, hence freezing the countdown for as long as necessary.

Display task
All processes need to output the state of the count or status information to the two 7-segment displays. As this involves parallel to serial conversion and shifting, the task is better gathered into one module.

Main process
The Main process is a loop displaying the Minute count until it reaches zero, with a premature break if the STOP switch is closed.

Set-time process
If the SET switch is closed when the PIC is reset then the SET_TIME subroutine quickly decrements the display count until the switch is released. This displayed value is then written into Data EEPROM and is used by all subsequent Main processes as the starting value for the Minute count.

Diagnostic process
If the DIAG switch is closed on reset, the system enters a diagnostic subroutine. The essentially exercises each peripheral device in a manner calculated to ease hardware fault finding.

All processes are dependent on the Timebase task to pass basic real-time clock information back. As shown in Program 16.1 this is interrupt driven and is based on the Timer 0:Prescaler dividing down the 3.2763 MHz crystal-driven oscillator to give overflow every $\frac{1}{50}$ s. As can be seen in Program 16.3, the Timer 0 interrupt is enabled and thus the PIC will enter ISR whenever the timer overflows – every 256 outputs from the Prescaler. Remembering that the crystal oscillator runs at $\frac{1}{4}$ of the crystal frequency, a prescale ratio of 1:64 will give a timebase rate of 50 per second – $\frac{3.2763\times10^6}{4\times64\times256} = 50$.

The task list for this function is:

1. IF PAUSE switch open THEN
   (a) Decrement the time chain by one Jiffy.
   (b) IF new second THEN flag it.
2. ELSE
   (a) Toggle the Pause flag.
   (b) IF set THEN tell the world that the system is paused.
   (c) ELSE display time to indicate normal running.
   (d) Wait until PAUSE switch is released.
3. Return from interrupt.

Program 16.1 The timebase software. (continued next page).

```
; ************************************************************
; * The ISR to decrement the real-time clock                 *
; * Adding a 20ms Jiffy on each entry                        *
; * Sets NEW_SEC to a non-zero value each Minute update      *
; ************************************************************
; First save context in usual way
ISR     movwf   _work       ; Put away W
        swapf   STATUS,w    ; and the Status register
        movwf   _status

; ************************************************************
; The core code
        btfss   INTCON,T0IF ; Was it a Timer0 time-out?
         goto   ISR_EXIT    ; IF no THEN false alarm

        btfsc   Pause,0     ; Check the Pause flag
         goto   ISR_EXIT    ; IF closed THEN don't increment

         bcf    INTCON,T0IF ; Clear interrupt flag
         incf   JIFFY,f     ; Record one more 1/50 second
        movlw   d'50'       ; Has Jiffy count reached 50?
        subwf   JIFFY,w
        btfss   STATUS,Z
         goto   ISR_EXIT    ; IF not THEN finished
        clrf    JIFFY       ; ELSE zero Jiffy count

        movf    SECOND,f    ; Test for Seconds count = 00?
        btfsc   STATUS,Z
         goto   NEW_MIN     ; IF it is THEN a NEW_SEC minute
        decf    SECOND,f    ; ELSE decrement Seconds count and
        incf    NEW_SEC,f   ; tell background prog new second
        goto    ISR_EXIT    ; and exit

NEW_MIN movlw   d'59'       ; Reset Seconds to 59 seconds
        movwf   SECOND
        movf    MINUTE,f    ; Test for Minutes count = 00?
        btfsc   STATUS,Z
         goto   ISR_EXIT    ; IF it is THEN no more decrement
        decf    MINUTE,f    ; ELSE decrement Minutes

 ; ************************************************************
ISR_EXIT btfss  PORTB,PAUSE ; Check the PAUSE switch
          call  FREEZE      ; IF closed THEN update Pause flag

         swapf  _status,w  ; Untwist the original Status reg
         movwf  STATUS
         swapf  _work,f    ; Get the original W reg back
         swapf  _work,w    ; leaving STATUS unchanged
         retfie             ; and return from interrupt
```

Program 16.1 (continued.) The timebase software.

```
; ************************************************************
; * FUNCTION: Increments the Pause flag.                     *
; * FUNCTION: IF = 1 THEN displays the decimal points        *
; * FUNCTION: IF = 0 THEN displays the normal count          *
; * RESOURCE: Subroutine SPI_WRITE. Var Pause                *
; * ENTRY   : PAUSE switch closed                            *
; * EXIT    : Pause switch open; appropriate display         *
; ************************************************************
FREEZE  incf     Pause,f     ; Update Pause flag, bit 0
        btfss    Pause,0     ; Check status of Pause flag
         goto    UNFREEZE    ; Change 1 -> 0, unfreeze
; Display freeze
        movlw    b'01111111' ; Code for display decimal point
        movwf    DATA_OUT_L
        movwf    DATA_OUT_H
        call     SPI_WRITE
        goto     FREEZE_EXIT

UNFREEZE ; Land here if Pause 0 -> 1.
        movf    MINUTE,w      ; Display the normal Minute count
        call    OUTPUT

FREEZE_EXIT
         btfss    PORTB,PAUSE  ; Wait til switch is opened again
          goto    FREEZE_EXIT
         return
```

From Program 16.1 we see that time is kept as a 3-byte count chain using file registers MINUTE, SECOND and JIFFY to hold the total. Assuming that the state of bit 0 of file register Pause is 0, then one is added to the Jiffy count. Normally the ISR then exits but when Jiffy reaches 50 it is reset to zero and the Seconds count decremented. The file register NEW_SEC is also made non zero to indicate to background software that a second has elapsed. In the situation where the Second count reaches zero then it is reset to 59 and the Minute count decremented. The procedure is similar to the incrementing count of Example 7.4.

The Timebase task also handles the Pause function. The simplest approach would be to skip over the time decrement code if the PAUSE switch is closed. However, the necessity to keep the switch closed could be irksome if the period was more than a few minutes.

Implementing a push-on push-off scenario is ergonomically superior and can be more economically implemented in software rather than using a different type of switch compared to the others. In Program 16.1 the Pause handling code is located in the separate subroutine FREEZE. It is permissible to call a subroutine from an ISR in the same manner as calling one subroutine from another; that is nesting. The 8-deep hardware stack

allows nesting up to eight deep. In our situation only two of the stack locations are used, allowing up to six calls deeper into the stack.

Subroutine FREEZE is only entered if the PAUSE switch is closed. On each entry the value of bit 0 of the file register Pause is toggled. This is implemented by simply incrementing file register Pause.

Once Pause[0] is toggled, its state is tested and if 1 the pattern to only illuminate the two decimal points is sent to the SPI_WRITE subroutine. This is an arbitrary indicator display, another possibility would be  . If Pause[0] is 0 then the state of the Minute count is sent to the OUTPUT subroutine and indicates to the user that the Pause function has ended.

Finally, the subroutine does not exit until the user releases the PAUSE switch. This is important, as on exit the ISR will be re-entered again at the next Timer 0 overflow, and this would cause Pause to be repeatedly retoggled. Some measure of switch debounce is obtained by zeroing Timer 0 and the Prescaler when the switch is released. This means that the switch will not be retested for a whole $\frac{1}{50}$ second. It is for this reason that T0IF is cleared on exit from the ISR rather than at the more conventional entry point.

The task displaying the contents of the Working register in decimal is handled by the subroutine OUTPUT in Program 16.2. The task list for this function is:

1. Convert the binary datum to 2-digit BCD.
2. Convert both digits to 7-segment.
3. Serially shift out both bytes to the appropriate display.

Program 16.2 The data display function. (continued next page).

```
; ************************************************************
; * FUNCTION: Displays datum as a 2-digit decimal output     *
; * RESOURCE: Subroutines BIN_2_BCD, SPI_WRITE, SVN_SEG      *
; * RESOURCE: Vars DATA_OUT_L, DATA_OUT_H, NEW_SEC, NUMBER   *
; * ENTRY   : Datum in W, <100d                              *
; * EXIT    : Data displayed, NEW_SEC zeroed                 *
; ************************************************************
OUTPUT   bcf    PORTA,SCK    ; Initialize the clock line
         call   BIN_2_BCD    ; Convert to BCD
         movwf  NUMBER       ; Put BCD MINUTE version in NUMBER

         movf   NUMBER,w     ; Get number count for display
         andlw  b'00001111'  ; Get Units nybble
         call   SVN_SEG      ; Convert to 7-segment code
         movwf  DATA_OUT_L   ; Copy into the serial low register
         swapf  NUMBER,w     ; Put ten's digit into lower nybble
         andlw  b'00001111'  ; Isolate ten's digit
         call   SVN_SEG      ; Convert to 7-segment code
         movwf  DATA_OUT_H   ; Copy into the serial high register
         call   SPI_WRITE    ; Shift both digits out

         clrf   NEW_SEC      ; Reset NEW_SEC flag
```

Program 16.2 (continued.) The data display function.

```
; ******************************************************
; * FUNCTION: Clocks out a two byte in parallel/series  *
; * ENTRY   : Data in DATA_OUT_L and DATA_OUT_H         *
; * ENTRY   : The former to be LSD, the latter MSD      *
; * EXIT    : DATA_OUT_L and DATA_OUT_H altered         *
; ******************************************************
SPI_WRITE
        bcf    PORTA,SCK     ; Make sure clock starts at low
        movlw  8             ; Initialize loop counter to 8
        movwf  COUNT
LOOP    bcf    PORTA,SDOH    ; Zero data bit for MSD
        rlf    DATA_OUT_H,f  ; Shift datum left into Carry
        btfsc  STATUS,C      ; Skip if Carry is 0
         bsf   PORTA,SDOH    ; ELSE make data bit 1
        bcf    PORTA,SDOL    ; Zero data bit for LSD
        rlf    DATA_OUT_L,f  ; Shift datum left into Carry
        btfsc  STATUS,C      ; Skip if Carry is 0
         bsf   PORTA,SDOL    ; ELSE make data bit 1
        bsf    PORTA,SCK     ; Pulse clock
        bcf    PORTA,SCK
        decfsz COUNT,f       ; Decrement count
         goto  LOOP          ; and repeat until zero
        return
SVN_SEG addwf PCL,f          ; Add N to PC giving PC + N
        retlw b'11000000'    ; Code for 0
        retlw b'11111001'    ; Code for 1
        retlw b'10100100'    ; Code for 2
        retlw b'10110000'    ; Code for 3
        retlw b'10011001'    ; Code for 4
        retlw b'10010010'    ; Code for 5
        retlw b'10000010'    ; Code for 6
        retlw b'11111000'    ; Code for 7
        retlw b'10000000'    ; Code for 8
        retlw b'10010000'    ; Code for 9
; ************************************************************
; * FUNCTION: Converts a binary byte to a packed BCD byte    *
; * RESOURCE: TEMP byte                                      *
; * ENTRY   : Binary byte in W range 00 - 63h (0 - 99d)      *
; * EXIT    : Packed BCD byte in W                           *
; ************************************************************
; Divide by ten
BIN_2_BCD clrf  TEMP         ; Zero the loop count
LOOP10    incf  TEMP,f       ; Record one ten subtracted
          addlw -d'10'       ; Subtract decimal ten
          btfsc STATUS,C     ; IF a borrow (C==0) THEN exit loop
           goto LOOP10       ; ELSE do another subtract/count
          decf  TEMP,f       ; Compensate for one inc too many
          addlw d'10'        ; Add ten to residue to give units
          swapf TEMP,f       ; Put ten's digit in upper nybble
          addwf TEMP,w       ; Add units nybble right justified
          return             ; and return to caller
```

Subroutine OUTPUT listed in Program 16.2 follows the task list calling up the following utility subroutines.

Binary to BCD conversion
Subroutine BIN_2_BCD repetitively subtracts ten from the binary datum in the manner described in Program 5.9 on page 131. Assuming that this datum is never greater than decimal 99 ($63h$) then this count gives the ten's digit. The residue is the unit's digit. The two nybbles are packed together and returned in W.

Binary to 7-segment decoder
Subroutine SVN_SEG converts a single datum nybble in W to its 7-segment coded equivalent as described in Program 6.4 on page 149.

SPI output
Subroutine SPI_WRITE is similar to that described in Program 12.1 on page 308 but transmits two serialized data streams simultaneously. The datum in DATA_OUT_L is sent via RA3 whilst that in DATA_OUT_H is sent out via RA0. A common clock is used.

Before considering the coding for the three processes, we will briefly look at the initialization code common to the entire software system. The function of this startup code is:

Default duration setting
To place a default value for the time-out in location 0 of the Data EEPROM. The address of this cell is in the special/test configuration area at $2100h$ and the de directive is used to specify the load time data as described on page 437.

The value of 10 as shown means that a freshly programmed PIC will default to a 10 minute count down. This value can subsequently be altered using the Set-time process described below.

Vectors
To initialize the Reset vector at $000h$ to point to MAIN and Interrupt vector at $004h$ to point to ISR.

Port setting
To make Port A[4:0] and Port B[7:5] outputs and all other lines inputs.

Timer 0 setting
To set up the Prescale ratio to 1:64 and Timer 0 clock source to internal. The Timer 0 interrupt is also enabled.

Process select
To check the state of the DIAG and SETT switches to choose either the Diagnostic or Set-time processes. If neither switch is closed the normal Main process is entered.

The Set switch is named SETT in the code, as set is a Microchip-compatible assembler directive.[4]

[4] set is the same as the equ directive except that the assigned value may be subsequently altered by other set directives.

Program 16.3 The initialization code.

```
          include  "p16f84.inc"
SDOH      equ 0
SCK       equ 1
BUZ       equ 2
SDOL      equ 3
GREEN     equ 5
YELLOW    equ 6
RED       equ 7
PAUSE     equ 0
DIAG      equ 1
STOP      equ 2
SETT      equ 3
GO        equ 4

          cblock   20h
           MINUTE:1, SECOND:1, JIFFY:1, NUMBER:1, NEW_SEC:1
           DATA_OUT_L:1, DATA_OUT_H, COUNT:1, TEMP:1, TIME_OUT:1
           Pause:1, _work:1, _status:1
          endc

          __config _XT_OSC & _WDT_OFF & _PWRTE_ON & _CP_OFF

          org     2100h       ; The EEPROM Data module
          de      d'10'       ; Default value is 10 minutes

RESET     org     0           ; Reset vector
          goto    MAIN
          org     4           ; Interrupt vector
          goto    ISR

MAIN      bsf     STATUS,RP0  ; Change to Bank 1
          movlw   b'11100000' ; RA4:0 outputs
          movwf   TRISA
          movlw   b'00011111' ; RB7:5 outputs; RB4:0 inputs
          movwf   TRISB
          movlw   b'00000101' ; Clock TMR0 internally; assigned PS
          movwf   OPTION_REG  ; Set to 1:64. Enable PORTB pull-ups
          bcf     STATUS,RP0  ; Back to Bank 0
          clrf    Pause       ; The PAUSE switch toggle
          clrf    NEW_SEC     ; Reset NEW_SEC second flag

          clrf    TMR0
          bcf     INTCON,T0IF
          bsf     INTCON,T0IE ; Enable Timer0 interrupts
          bsf     INTCON,GIE  ; Enable all interrupts

          btfss   PORTB,SETT  ; Check the Set switch
           call   SET_TIME    ; IF closed THEN set total time
          btfss   PORTB,DIAG  ; Check the Diagnostic switch
           call   DIAGNOSTIC  ; IF closed THEN set total time
```

If the DIAG switch is closed when the PIC comes out of reset then the code transfers to the subroutine DIAGNOSTIC.

The Diagnostic process aims to exercise the various peripheral devices interfaced to the process in order to verify in a reproducible manner the status of the interconnection and the devices themselves.

Switches

Five switches are input via Port B. By checking each switch in turn and if closed lighting one of the LEDs or sounding the buzzer both switches and the listed output devices are tested. The DIAG switch is of course verified by moving the system into this process and the Reset switch is tested by initiating the startup process.

If there were more switches than output devices then either combinations of the latter could be activated or else one or more segments in the numerical display pushed into service.

Program 16.4 The Diagnostic process.

```
; ************************************************************
; * FUNCTION: Checks each switch and activates a corresponding*
; * FUNCTION: LED or buzzer. Continually activates a unary    *
; * FUNCTION: pattern to both 7-segment displays              *
; * RESOURCE: Subroutines SPI_WRITE                           *
; * RESOURCE: Vars TEMP, DATA_OUT_H, DATA_OUT_L               *
; * ENTRY   : DIAG switch closed                              *
; * EXIT    : DIAG switch open                                *
; ************************************************************
DIAGNOSTIC
         movlw  b'11111110'  ; The initial 7-segment pattern
         movwf  TEMP         ; in memory
D_LOOP   movlw  b'11111111'  ; Turn off all LEDs and buzzer
         movwf  PORTB
         bsf    PORTA,BUZ
; Now scan switches
         btfss  PORTB,PAUSE  ; IF Pause switch closed
          bcf   PORTB,GREEN  ; THEN Green LED
         btfss  PORTB,STOP   ; IF Stop switch closed
          bcf   PORTB,YELLOW ; THEN Yellow LED
         btfss  PORTB,SETT   ; IF Set switch closed
          bcf   PORTB,RED    ; THEN Red LED
         btfss  PORTB,GO     ; IF Go switch closed
          bcf   PORTA,BUZ    ; THEN Buzzer
; Now turn on each segment in turn of both displays
         movf   TEMP,w       ; Get pattern
         movwf  DATA_OUT_L   ; Put in output file regs
         movwf  DATA_OUT_H
         call   SPI_WRITE    ; Display it

         btfsc  PORTB,DIAG   ; IF Diagnostic switch open
         return              ; THEN exit the diag subroutine
         clrf   NEW_SEC      ; Reset the New Second flag
; Now move the display pattern on one and wait for a second
         bcf    STATUS,C     ; Clear Carry
         btfsc  TEMP,7       ; Check MSB of pattern
          bsf   STATUS,C     ; IF 1 THEN Carry = 1
         rlf    TEMP,f       ; Shift it in <<

D_LOOP2  movf   NEW_SEC,f    ; ELSE wait for the new second
         btfsc  STATUS,Z     ; IF non zero THEN skip
          goto  D_LOOP2      ; ELSE try again
         goto   D_LOOP       ; Repeat routine
```

LEDs and buzzer
The static output devices are tested in conjunction with the switch test listed above. Of course the failure of a LED to light or buzzer to sound may be due to either the input or output device circuit. Determining which is easily accomplished by using a voltmeter or logic probe. Also all LEDs are illuminated during the Set-time process.

Display
Each of the display devices is exercised by lighting one segment moving on once each second in an endless loop. This is implemented by generating a walking unary pattern 11111110 → 11111101 → · · · 01111111 sent out to the output subroutine SPI_WRITE once each time the file register NEW_SEC is non zero. NEW_SEC is incremented in the Timer 0 interrupt-handling routine each time the Seconds count is incremented and cleared in the Diagnostic procedure code. This acts as a ratchet giving only one new display each second.

The Set-time process is entered when the SETT switch is closed whenever the processor comes out of reset. Its function is to allow the operator to change the contents of the EEPROM Data module location $00h$ to any value up to 99. This location holds the initial count-down value used by the Main process to determine the length of the procedure.

The strategy behind the coding shown in Program 16.5 is to initialize the Second count to 99 and let it decrement at a 1-second rate as

Program 16.5 The Set-time process.

```
; ************************************************************
; * FUNCTION: Slowly counts down from 99-00. When Set switch  *
; * FUNCTION: released EEPROM is Written with NEW_SEC time-out*
; * RESOURCE: Subroutines DISPLAY, EE_PUT, ISR; Var TIME_OUT  *
; * ENTRY   : Set switch is closed                            *
; * EXIT    : EEPROM Data address 00 is updated               *
; ************************************************************
SET_TIME movlw  d'99'        ; Start count at 99 seconds
         movwf  SECOND
         movlw  b'00000000'  ; All LEDs on
         movwf  PORTB
SET_LOOP movf   SECOND,w     ; Get Second count
         call   OUTPUT       ; Display it and clear NEW_SEC
         btfsc  PORTB,SETT   ; Check; does the user want to stop?
          goto  UPDATE       ; IF yes THEN update EEPROM and exit
         movf   SECOND,w     ; Get displayed count
         movwf  TIME_OUT     ; Make a temporary copy
S_LOOP   movf   NEW_SEC,f    ; Check NEW_SEC status
          btfsc STATUS,Z     ; IF non zero THEN skip
         goto   S_LOOP       ; ELSE try again
         goto   SET_LOOP     ; Repeat display
UPDATE   movf   TIME_OUT,w   ; Get the value
         movwf  EEDATA       ; Set up EEPROM
         clrf   EEADR
         call   EE_PUT       ; Program EEPROM
         return              ; and return to main program
```

determined by the foreground ISR. The value of SECOND is sent to the Display subroutine each time the ISR sets the flag file register NEW_SEC to a non zero value, that is once per second. DISPLAY clears NEW_SEC so the net effect to update the display only once each second. Each second the SETT switch is checked and when open the state of the Seconds count is transferred to the EEPROM Data module at UPDATE using the EE_PUT subroutine of Program 15.2 on page 436.

The Complete background system flow chart is shown in Fig. 16.3. This shows in outline the decision flow taken after a reset and in detail

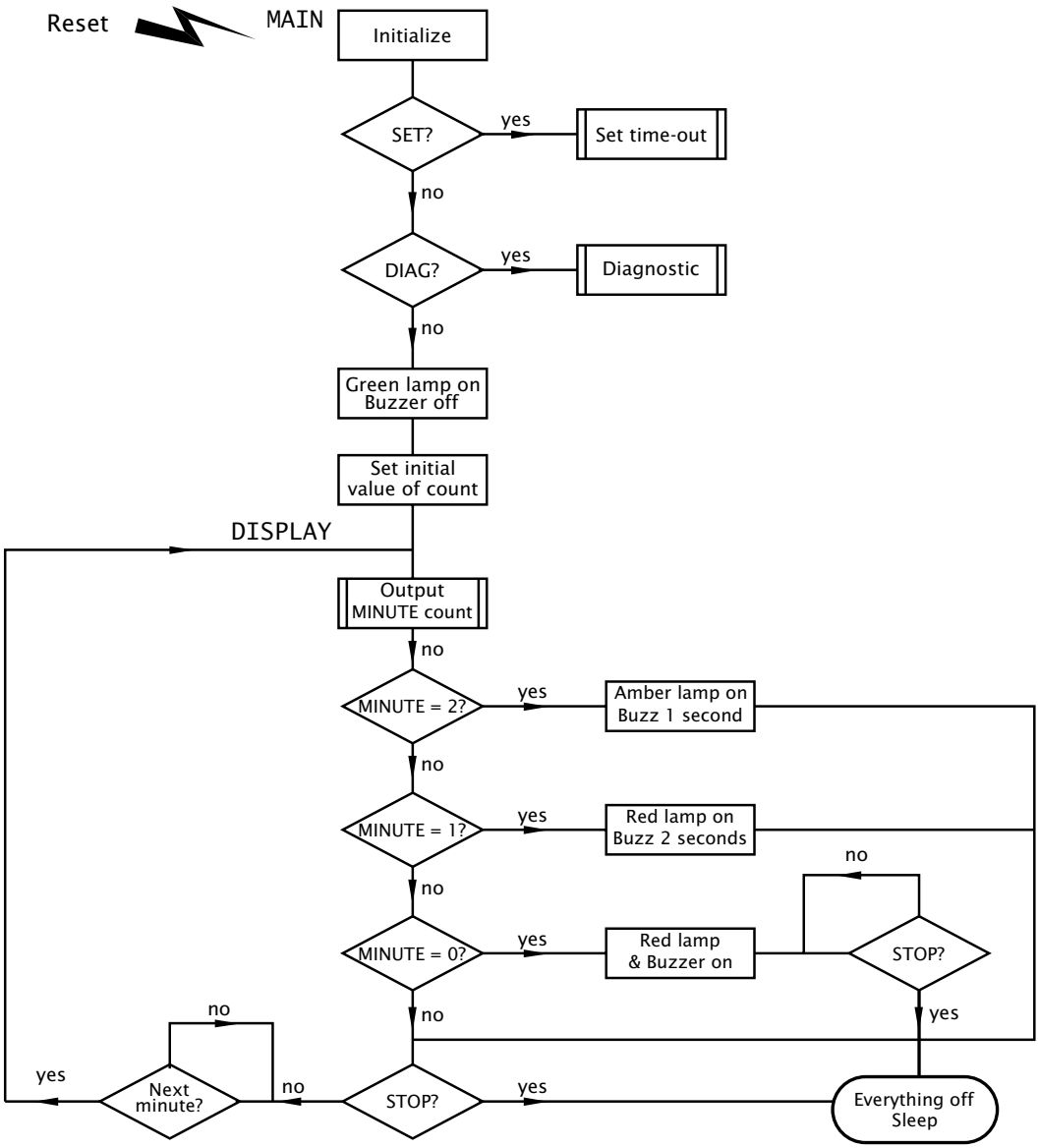


Fig. 16.3 The Main process.

the Main process. Although this looks rather complex, it may be broken down into five phases with corresponding coding shown in Program 16.6.

Preamble
On reset if neither SETT or DIAG switches are closed the Main procedure code is entered at MAIN_PROC. This reads the initial value of the count-down period from EEPROM location 00*h* and initializes the count chain. The green lamp is illuminated and other lamps and buzzer are turned off.

Countdown
The Countdown phase continually displays the Minute count – updated behind the scenes by the ISR. The green lamp remains illuminated as long as this display does not drop below 03. This phase is complete whenever the count drops below 3 minutes or else the STOP switch is closed. In the latter case all displays are blanked and the PIC is put into its Sleep state.

In all situations except where the STOP command is issued the Minute count is displayed at 1-minute intervals. The routine at REPEAT checks the Second count and if zero the loop is repeated – that is once per minute. The simpler alternative of continually refreshing the 7-segment readouts gives an inferior display as the data being serially shifted at frequent rate may partially illuminate segments which are nominally off. In addition, repeating the loop each minute eases the task of sounding the buzzer once only when the Minute count drops to two and one.

Two minutes to go
When the display is 02 the amber lamp is illuminated This is timed using the NEW_SEC variable. Again the loop can be prematurely exited if the STOP switch has been closed.

One minute to go
When the display is 01 the loop diverts to illuminate the red lamp. The buzzer is sounded for two seconds; implemented in code as two 1-second buzzes.

Timed out
When the Minute count reaches zero, not only is 00 displayed but also the buzzer sounds continually. This cacophony can only be silenced by pressing the Stop switch – or by resetting and starting again. As in previous situations when the Stop switch is closed, all displays are blanked out and the PIC is placed in its Sleep state.

Once the source code has been assembled and where possible simulated (see Fig. 8.7 on page 222) it can then be burnt into the PIC's Program store. In the first instance only the diagnostic software and associated tasks need be programmed in order to check the target hardware. The precise details will depend somewhat on the PIC programmer being used and its associated software.

Program 16.6 The Main process. (continued next page).

```
         movlw  b'11000000'  ; Green LED on
         movwf  PORTB
         bsf    PORTA,BUZ    ; Buzzer off

; Get start value from EEPROM

         clrf   EEADR        ; EEPROM address zero
         call   EE_GET       ; Get the start value
         movwf  MINUTE
         movlw  d'59'        ; Initial value for seconds
         movwf  SECOND       ; is 59
         clrf   JIFFY

DISPLAY  movf   MINUTE,w     ; Get Minute count
         call   OUTPUT       ; Output to display

; The 2-minutes-to-go phase ***********************************
; At a count of two sound the buzzer for one second and turn on
; the amber lamp
TWO      movf   MINUTE,w     ; Minute count = 2?
         addlw  -2
         btfss  STATUS,Z
          goto  ONE          ; IF not THEN try for one minute
         movlw  b'10100000'  ; Amber LED on
         movwf  PORTB
         bcf    PORTA,BUZ    ; Buzzer on
TWO_LOOP movf   NEW_SEC,f    ; Check NEW_SEC status
         btfsc STATUS,Z      ; IF non zero THEN skip
          goto  TWO_LOOP     ; ELSE try again
         bsf    PORTA,BUZ    ; Turn off buzzer after one second
         goto   REPEAT       ; repeat display

; The 1-minute-to-go phase ************************************
; At a count of one sound the buzzer for two second and turn on
; the red lamp
ONE      movf   MINUTE,w     ; Minute count = 1?
         addlw  -1
         btfss  STATUS,Z
          goto  ZERO         ; IF not THEN try for zero minutes
         movlw  b'01100000'  ; Red LED on
         movwf  PORTB
         bcf    PORTA,BUZ    ; Buzzer on
ONE_LOOP movf   NEW_SEC,f    ; Check NEW_SEC status
         btfsc STATUS,Z      ; IF non zero THEN skip
          goto  ONE_LOOP     ; ELSE try again
         clrf   NEW_SEC      ; Again clear NEW_SEC flag
UN_LOOP  movf   NEW_SEC,f    ; Again check NEW_SEC status
         btfsc STATUS,Z      ; IF non zero THEN skip
          goto  UN_LOOP      ; ELSE try again
         bsf    PORTA,BUZ    ; Turn off buzzer after two seconds
         goto   REPEAT       ; Repeat display
```

Program 16.6 (continued.) The Main process

```
; The Timed-Out phase ****************************************
; When the Minute count reaches zero, sound the buzzer
; until the Stop switch is closed
ZERO     movf   MINUTE,f     ; Minute count = 0?
         btfss  STATUS,Z
          goto  REPEAT       ; IF not THEN repeat after minute
         bcf    PORTA,BUZ    ; Buzzer on
ZERO_LOOP
         btfsc  PORTB,STOP   ; Check the Stop switch
          goto  ZERO_LOOP    ; and continue until closed
FINI     movlw  b'11100000'  ; Turn lamps off
         movwf  PORTB
         bsf    PORTA,BUZ    ; and buzzer
         movlw  b'11111111'  ; Code for blank
         movwf  DATA_OUT_L
         movwf  DATA_OUT_H
         call   SPI_WRITE    ; Blank both displays
         sleep               ; and await another reset

REPEAT   btfss  PORTB,STOP   ; Check the Stop switch
          goto  FINI         ; IF closed THEN freeze
         movf   SECOND,f     ; Wait until Second count is again zero
          btfss STATUS,Z     ; i.e. for the next minute
         goto   REPEAT       ; IF not THEN wait again
         clrf   NEW_SEC      ; ELSE wait one more second
R_LOOP   movf   NEW_SEC,f    ; Check NEW_SEC status
         btfsc  STATUS,Z     ; IF non zero THEN skip
          goto  R_LOOP       ; ELSE try again
         goto   DISPLAY      ; Repeat display
```

The screen shot shown in Fig. 16.4 shows the situation where the Microchip Picstart Plus development programmer is used in conjunction with the MPLAB IDE. Communication with the host computer is via a RS-232 serial port and contact is made from the Picstart Plus menu. The right-hand window allows the operator to set up the Configuration Bits (fuses), shown in the left-hand window. Once this is set up, the operator can Blank out, Read from, Program or Verify the contents of the EPROM or EEPROM Program store is the same as that produced by the last assembly process. This process can only be carried out where the Code Protect has not been turned on. Once this is the case, it is irreversible and neither Program or Verify tasks can be carried out.

The middle window shows the status of the Program or Verify process. As shown here, it is announcing that it has completed the task up to 03FF*h* and is reporting success. The complete process takes less than a

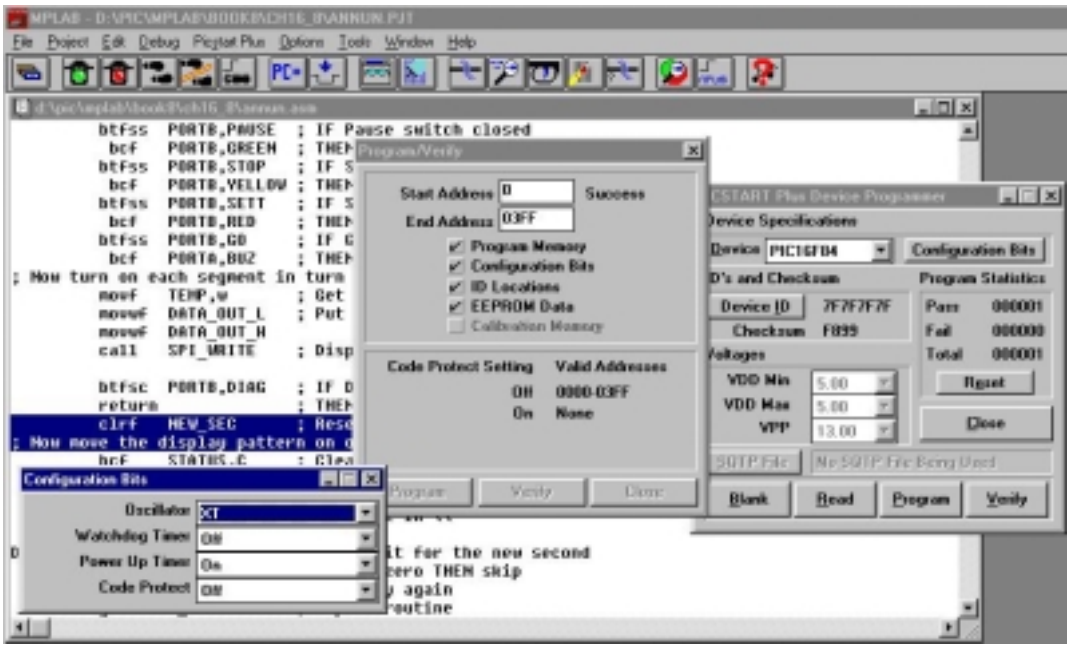


Fig. 16.4 Programming the PIC from MPLAB.

minute for the PIC16F84 and the 252 program words that this case study software generates.[5]

With F series PICs, the programming process may be repeated without any preparation of the Program store up to 100 times without any deterioration of the Flash EEPROM. C series PICS,[6] such as the PIC16C74, have EPROM Program stores. Where the target has a quartz window it must be erased using a suitable UV-based EPROM eraser for approximately 20 minutes before programming, unless code is being added to previously unprogrammed memory. Although quartz windowed devices are necessary for development purposes, they are relatively expensive. Thus cheaper windowless version are used for production purposes, and are called One-Time Programmable (OTP) since they cannot subsequently be erased. Part numbers which are windowed are usually identified by JW postfix; for example, PIC16C74B-20/JW is a 20 MHz ceramic windowed PIC16C74B part and and the PIC16C74B-4/P is a 4 MHz OTP version in a 40-pin plastic DIL package. Ensure that you obtain the correct device!

The hardware and software circuits have been presented here as a simple illustrative case study to integrate many of the techniques described in the body of the text. If you decide to build your own version,files, **C** coding, PCB, comparison with a Motorola 68000 MPU version and other ideas

[5] With 772 instructions left unused, the PIC18F83 with a Program store of 512 words could be used as the target process with a small reduction in cost.

[6] The exception being the obsolete PIC16C83/4 which also has an EEPROM Program store.

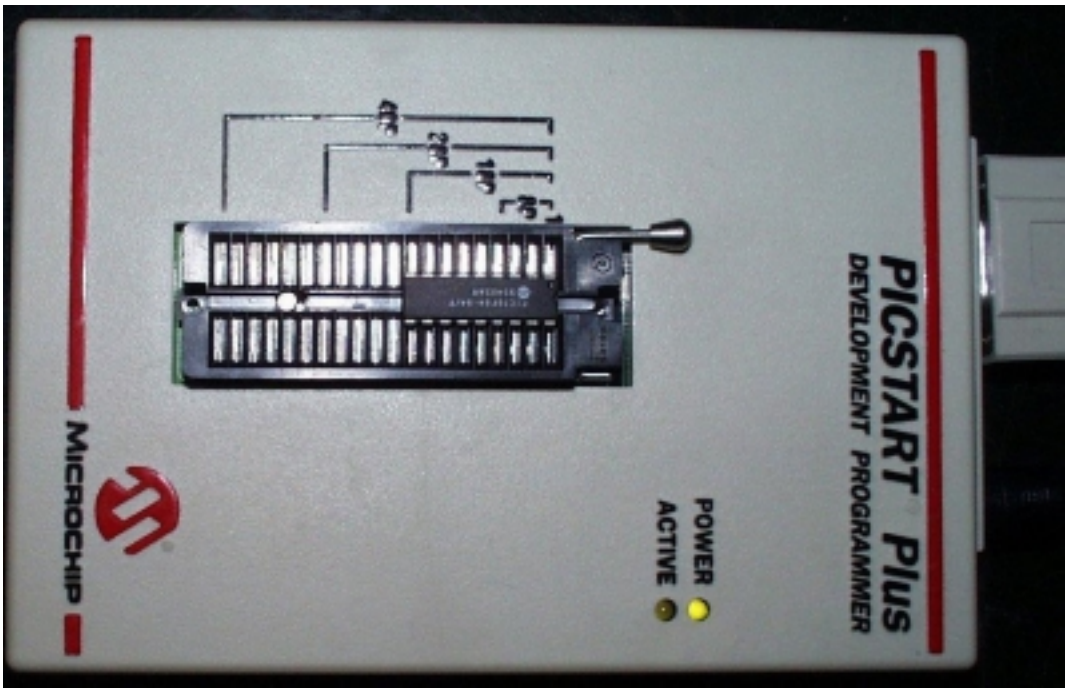
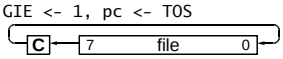
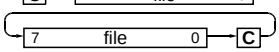


Fig. 16.5 The Microchip PICSTART Plus programmer.

for experimentation, which you are welcome to contribute are given on the associated Web site detailed in the Introduction. Good luck!

Appendix A

14-bit Core Instruction Set

14-bit op-code	16CPX 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 == 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 == 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; 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 <- w - f
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	==	: Equivalent to
pc++	: Jump over next instruction	++	: Add one	--	: Subtract one
GIE	: Global Interrupt Enable mask	#	: Constant number		

Appendix B

Special Purpose Register Structure for the PIC16C74B

File	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	—	—	RA5	RA4	RA3	RA2	RA1	RA0	—XX XXXX	—-UU UUUU
06*h*	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0	XXXX XXXX	UUUU UUUU
07*h*	PORTC	RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0	XXXX XXXX	UUUU UUUU
08*h*	PORTD	RD7	RD6	RD5	RD4	RD3	RD2	RD1	RD0	XXXX XXXX	UUUU UUUU
09*h*	PORTE	—	—	—	—	—	RE2	RE1	RE0	—— —XXX	—— —UUU
0A*h*	PCLATH	—	—	—	Write buffer for top 5 PC bits					——0 0000	——0 0000
0B*h*	INTCON	GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000X	0000 000U
0C*h*	PIR1	PSPIF	ADIF	RCIF	TXIF	SSPIF	CCPIF	TMR2IF	TMR1IF	0000 0000	0000 0000
0D*h*	PIR2	—	—	—	—	—	—	—	CCP2IF	—— ——0	—— ——0
0E*h*	TMR1L	Timer 1 Low Byte								XXXX XXXX	UUUU UUUU
0F*h*	TMR1H	Timer 1 High Byte								XXXX XXXX	UUUU UUUU
10*h*	T1CON	—	—	T1CKPS1	TICKPS0	T1OSCEN	$\overline{T1SYNC}$	TMRCS	TMR1ON	—00 0000	—UU UUUU
11*h*	TMR2	Timer 2								0000 0000	0000 0000
12*h*	T2CON	—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0	-000 0000	-000 0000
13*h*	SSPBUF	Synchronous Serial Port Receive Buffer/ Transmit Register								XXXX XXXX	UUUU UUUU
14*h*	SSPCON	WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1S1	SSPM0	00000 0000	00000 0000
15*h*	CCPR1L	Capture/Compare/PWM register 1 Low byte								XXXX XXXX	UUUU UUUU
16*h*	CCPR1H	Capture/Compare/PWM register 1 High byte								XXXX XXXX	UUUU UUUU
17*h*	CCP1CON	—	—	CCP1X	CCP1Y	CCP1M3	CCP1M2	CCP1M1	CCP1M0	—00 0000	—00 0000
18*h*	RCSTA	SPEN	RX9	SREN	CREN	—	FERR	OERR	RXD8	0000 -00X	0000 -00X
19*h*	TXREG	SCI Transmit Data Register								0000 0000	0000 0000
1A*h*	RCREG	SCI Receive Data Register								0000 0000	0000 0000
1B*h*	CCPR2L	Capture/Compare/PWM register 2 Low byte								XXXX XXXX	UUUU UUUU
1C*h*	CCPR2H	Capture/Compare/PWM register 2 High byte								XXXX XXXX	UUUU UUUU
1D*h*	CCP2CON	—	—	CCP2X	CCP2Y	CCP2M3	CCP2M2	CCP2M1	CCP2M0	—00 0000	—00 0000
1E*h*	ADRES	A/D Result Register								XXXX XXXX	UUUU UUUU
1F*h*	ADCON0	ADCS1	ADCS0	CHS2	CHS1	CHS0	GO	—	ADON	0000 00-0	0000 00-0

X Not known
U Unchanged
? Value depends on reset condition.
— Unimplemented; read as 0.
Note 1: Next instruction address if PIC in Sleep mode.

File address	Name	7	6	5	4	3	2	1	0	Power-on Reset	All other Resets
Bank 1											
80*h*	INDF	Uses contents of this to address Data memory (not a physical register)									
81*h*	OPTION	$\overline{\text{RBPU}}$	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	$\overline{\text{TO}}$	$\overline{\text{PD}}$	**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						—-11 1111	—-11 1111
86*h*	TRISB	Port B Data Direction Register								1111 1111	1111 1111
87*h*	TRISC	Port C Data Direction Register								1111 1111	1111 1111
88*h*	TRISD	Port D Data Direction Register								1111 1111	1111 1111
89*h*	TRISE	IBF	OBF	IBOV	PSPM	—	TRISE2	TRISE1	TRISE0	0000 -111	0000 -111
8A*h*	PCLATH	—	—	—	Write buffer for top 5 PC bits					——0 0000	——0 0000
8B*h*	INTCON	GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000X	0000 000U
8C*h*	PIE1	PSPIE	ADIE	RCIE	TXIE	SSPIE	CCPIE	TMR2IE	TMR1IE	0000 0000	0000 0000
8D*h*	PIE2	—	—	—	—	—	—	—	CCP2IE	—— ——0	—— ——0
8E*h*	PCON	—	—	—	—	—	—	$\overline{\text{POR}}$	$\overline{\text{BOR}}$	—— ——??	—— ——UU
92*h*	PR2	Timer 2 Period Register								1111 1111	1111 1111
93*h*	SSPADD	Synchronous Serial Port (I^2C mode) Address Register								0000 0000	0000 0000
94*h*	SSPSTAT	—	—	D/$\overline{\text{A}}$	P	S	R/$\overline{\text{W}}$	UA	BF	—-00 0000	—-00 0000
98*h*	TXSTA	CSRC	TX9	TXEN	SYNC	—	BRGH	TRMT	TX9D	0000 -010	0000 -010
99*h*	SPBRG	Baud Rate Generator								0000 0000	0000 0000
9F*h*	ADCON1	—	—	—	—	—	PCFG2	PCFG1	PCFG0	—— —000	—— —000

X Not known
U Unchanged
? Value depends on reset condition.
— Unimplemented; read as 0.
Note 1: Next instruction address if PIC in Sleep mode.

Appendix C

C Instruction Set

Operator	Operation	Example
Top priority		
Direction (associativity) ⇒		
`()`	Function call	`sqr()`
`[]`	Array element	`x[6]`
`.`	Structure element	`PIA1.CRA`
`->`	Structure element using a pointer	
Unary operators		
Direction (associativity) ⇐		
`!`	Logical NOT	`!x`
`~`	Inversion (1's complement)	`~x`
`-`	Negative	`y=-x`
`+`	Unary plus	`y=x- +(y+z)`
`++`	Increment	`x++` or `++x`
`--`	Decrement	`x--` or `--x`
`&`	Address of	`&x`
`*`	Contents of address	`*address`
`(type)`	Cast	`(long)x`
`sizeof`	Size of object in bytes	`sizeof x`
Arithmetic		
Direction (associativity) ⇒		
`*`	Multiplication	`z=x*y`
`/`	Division	`z=x/y`
`%`	Remainder	`z=x%y` (Integer types only)
`+`	Addition	`z=x+y`
`-`	Subtraction	`z=x-y`
Shift		Integer types only
Direction (associativity) ⇒		
`>>`	Shift left	`z=x>>3`
`<<`	Shift right	`z=x<<3`
Relational operators		Boolean objects
Direction (associativity) ⇒		
`<`	Less than	`while (x<3 )`
`<=`	Less than or equal	`while (x<=3)`
`>`	Greater than	`while (x>3 )`
`>=`	Greater than or equal	`while (x>=3)`
`==`	Equivalent	`while (x==y)`
`!=`	Not equivalent	`while (x!=0)`

C operators, their precedence and associativity (continued next page).

Operator	Operation	Example
Bitwise logic		Integer types only
Direction (associativity) ⇒		
&	AND	x&0xFE (Clear bit 0)
^	Exclusive-OR	x^0x01 (Toggle bit 0)
|	OR	x|0x01 (Set bit 0)
Objectwise logic		Boolean objects
Direction (associativity) ⇒		
&&	Logical AND	x&&y is True if both x and y are True
||	Logical OR	x||y is True if both or either x and y are True
?:	Conditional	x=(y>z)?5:10 x=5 if y>z True else x=10
Assignment		
Direction (associativity) ⇐		
=	Simple	x=3
+=	Compound plus	x+=3 e.g. (x=x+3)
-=	Compound minus	x-=3 e.g. (x=x-3)
*=	Compound multiply	x*=3 e.g. (x=x*3)
/=	Compound divide	x/=3 e.g. (x=x/3)
%=	Compound remainder	x%=3 e.g. (x=x%3)
&=	Compound bit AND	x&=3 e.g. (x=x&3)
^=	Compound bit EX-OR	x^=3 e.g. (x=x^3)
|=	Compound bit OR	x|=3 e.g. (x=x|3)
<<=	Compound shift left	x<<=3 e.g. (x=x<<3)
>>=	Compound shift right	x>>=3 e.g. (x=x>>3)
Direction (associativity) ⇒		
,	Concatenate	if(x=0,y=3;x<10,x++)
Lowest priority		

(continued.) **C** operators, their precedence and associativity.

Appendix D

Acronyms and Abbreviations

ADC (A/D)	Analog to Digital converter/conversion
ADCON0	A/D CONtrol 0; File $08h$ (PIC16C71)
ADCON1	A/D CONtrol 1; File $88h$ (PIC16C71)
ADCSn	A/D Clock Select, bits 1:0 in the ADCON0 file register
ADIE	A/D Interrupt Enable mask; INTCON[6] (PIC16C71X)/PIE1[6]
ADIF	A/D Interrupt Flag; ADCON0[1] (PIC16C71X)/PIR1[6]
ADON	A/D module ON; ADCON0[0]
ADRES	A/D RESult; File $09h/89h$ (PIC16C71X)/File $1Eh$
ALU	Arithmetic Logic Unit
ANn	A/D input pin n
ANSII	American National Standards Institution
ALU	Arithmetic Logic Unit
ASCII	American Standard Code for Information Interchange
BCD	Binary Coded Decimal
BF	Buffer Full; SSPSTAT[0]
C	Carry flag in the Status register; STATUS[0]
CCP	Capture/Compare PWM module
CCP1	CCP1 input/output pin (Common with RC2)
CCPR1H	CCP Register 1 High byte; File $15h$
CCPR1L	CCP Register 1 Low byte; File $16h$
CCP1CON	CCP1 CONtrol register; File $1Dh$
CCP1IE	CCP1 Interrupt Enable mask; PIE1[2]
CCP1IF	CCP1 Interrupt Flag; PIR1[2]
CCP2IE	CCP2 Interrupt Enable mask; PIE2[0]
CCP2IF	CCP2 Interrupt Flag; PIR2[0]
CCP1M	CCP Mode control; CCP1CON[3:0]
CHSn	A/D CHannel Select; ADCON[5:3]
CISC	Complex Instruction Set Computer
CKP	ClocK Polarity; SSPCON[4]
CMOS	Complimentary Metal-Oxide Semiconductor
CREN	Continuous Receive ENable; RCSTA[4]
$\overline{CS}$	Chip Select
CTS	Clear To Send, RS-232 handshake signal
DAC (D/A)	Digital to Analog converter/conversion
DC	Digit Carry flag in the Status register; STATUS[1]
DC1B	Duty Cycle 1 Bits; CCP1CON[5:4]
DCE	Data Circuit terminating Equipment
DTE	Data Terminal Equipment
ea	Effective Address
EEADR	EEPROM ADdress; File $09h$ or File $10Dh$
EEADRH	EEPROM ADdress High byte; File $10Fh$

EECON1	EEPROM CONtrol 1; File 88h or File 18Ch
EECON2	EEPROM CONtrol 2; File 89h or File 18Dh
EEDATA	EEPROM DATA; File 08h or File 10Ch
EEDATH	EEPROM DATa High byte; File 10Eh
EEIE	EEPROM Interrupt Enable mask; INTCON[6] or PIE2[4]
EEIF	EEPROM Interrupt Flag; EECON1[4] or PIR2[4]
EEPGD	EEPROM ProGram/$\overline{\text{Data}}$; EECON1[7].
EEPROM	Electrical Erasable PROM
EPROM	Erasable PROM
FERR	Framing ERRor; RCSTA[2]
FSR	File Select Register; File 4
GIE	Global Interrupt Enable mask; INTCON[7]
GO/$\overline{\text{DONE}}$	ADC Start Convert (GO)/End Of Conversion (DONE); ADCON0[2]
GPR	General-Purpose file Register
GPn	General-purpose Register (I/O port) GPIO pin n
IC	Integrated Circuit
I^2C	Inter-Integrated Circuit serial protocol
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
INDF	INDirect File register; File 0
INT	External INTerrupt input pin (Common with RB0)
INTCON	INTerrupt CONtrol Register; File 0Bh
INTEDG	External INTerrupt EDGe polarity selection; OPTION_REG[0]
INTE	INTerrupt Enable; INTCON[4]
INTF	INTerrupt Flag; INTCON[1]
I/O	Input/Output
IRP	Indirect addressing Register Page; STATUS[7]
ISR	Interrupt Service Routine
LED	Light-Emitting Diode
LSB	Least Significant Bit or Byte
LSI	Large Scale Integration
LSD	Least Significant Digit
$\overline{\text{MCLR}}$	Master CLeaR reset pin
MCU	MicroController Unit
MPU	MicroProcessor Unit
ms	Millisecond (10^{-3} s)
MSB	Most Significant Bit or Byte
MSD	Most Significant Digit
MSI	Medium Scale Integration
ns	Nanosecond (10^{-9} s)
$\overline{\text{OE}}$	Output Enable
OERR	Overflow ERRor; RCSTA[1]
Operating System	OS
OPTION_REG	OPTION file REGister; File 81h
OTP	One-Time Programmable (EPROM)
PC	Program Counter
PC	Personal Computer
PCFGn	A/D Port ConFiGuration; ADCON1[2:0]
PCL	Program Counter Low byte; File 2
PCLATH	Program Counter LATch High byte; File 0Ah
$\overline{\text{PD}}$	Power Down sleep mode; STATUS[3]
PEIE	PEripheral Interrupt Enable mask; INTCON[6]

PIC	Peripheral Interface Controller
PIPO	Parallel-In Parallel-Out register
PIE1	Peripheral Interrupt Enable register 1, File 8Ch
PIE2	Peripheral Interrupt Enable register 2, File 8Dh
PIR1	Peripheral Interrupt Register 1, File 0Ch
PIR2	Peripheral Interrupt Register 2, File 0Dh
PISO	Parallel-In Serial-Out shift register
PR2	Period Register for Timer 2; File 92h
PRNG	Pseudo Random Number Generator
PROM	Programmable ROM
PS	Post/Prescale rate Select; OPTION_REG[2:0]
PSA	Post/Prescale Scaler Assign; OPTION_REG[3]
PWM	Pulse Width Modulation
RAn	Register (port) A I/O pin n
RAM	Random Access Memory
RBn	Register (port) B I/O pin n
RBIE	Register B Interrupt Enable; INTCON[3]
RBIF	Register B Interrupt Flag; INTCON[0]
$\overline{\text{RBPU}}$	Register B Pull-UP; OPTION_REG[7]
RCIE	ReCeive register Interrupt Enable mask; PIE1[5]
RCIF	ReCeive register Interrupt Flag; PIR1[5]
RCREG	ReCeive data REGister; File 1Ah
RCSTA	ReCeive STAtus register; File 18h
RD	ReaD; EECON1[0] file register
REn	Register (port) E I/O pin n
RISC	Reduced Instruction Set Computer (see CISC)
ROM	Read-Only Memory
RPn	Register Page bits; STATUS[6:5]
rtl	Register Transfer Language
RTS	Ready To Send, RS-232 handshake signal
RX	ReCeive pin for USART (common with RC7)
RX9	ReCeive 9-bit data control; RCSTA[6]
RTCC	Real Time Counter/Clock (see Timer 0)
SAR	Successive Approximation Register
SCK	Serial ClocK in SPI protocol
SDI	Serial Data Input in SPI protocol
SDO	Serial Data Output in SPI protocol
SIPO	Serial-In Parallel-Out shift register
SISO	Serial-In Serial-Out shift register
SP	Stack Pointer
SPBRG	Serial Port Baud-Rate Generator; File 99h
SPEN	Serial Port ENable; RCSTA[7]
SPI	Serial Peripheral Interface protocol
SPR	Special-Purpose file Register
SSP	Synchronous Serial Port
SSPBUF	SSP BUFfer register; File 13h (PIC16C7X)
SSPCON	SSP CONtrol file REGister, File 14h
SSPEN	SSP Enable; SSPCON[5]
SSPIE	SSP Interrupt Enable mask; PIE1[3]
SSPIF	SSP Interrupt Flag; PIR1[3]
SSPM	SSP Mode control; SSPCON[3:0]
SSPSR	SSP Shift Register (PIC16C7X)
STATUS	Status Register; File 3

SYNC	SYNChronous mode in the USART; TXSTA[4]
$\overline{\text{TO}}$	Watchdog Time Out; STATUS[4]
T0CKI	Timer 0 ClocK Input pin (Common with RA4)
T0CS	Timer 0 Clock Select; OPTION_REG[5]
T0IE	Timer 0 Interrupt Enable mask; INTCON[5]
T0IF	Timer 0 Interrupt Flag; INTCON[2]
T1CKI	Timer 1 ClocK Input pin (Common with RC0)
T1CON	Timer 1 CONtrol register; File $10h$
T2CON	Timer 2 CONtrol register; File $12h$
T1OSCEN	Timer 1 OSCillator ENable; T1CON[3]
$\overline{\text{T1SYNC}}$	Timer 1 SYNChronize; T1CON[2]
TMR0	TiMeR 0 file register; File 1
TMR1CS	TiMeR 1 Clock Select; T1CON[1]
TMR1H	TiMeR 1 High byte file register; File $0Fh$
TMR1IE	Timer 1 Interrupt Enable mask; PIE1[0]
TMR2IE	Timer 2 Interrupt Enable mask; PIE1[1]
TMR1IF	Timer 1 Interrupt Flag; PIR1[0]
TMR2IF	Timer 2 Interrupt Flag; PIR1[1]
TMR1L	TiMeR 1 Low byte file register; File $0Eh$
TMR1ON	TiMeR 1 ON; T1CON[0]
TOUTPS	Timer 2 OUTput Post Scaler; T2CON[3:0]
TRISA	TRIState A (Data Direction register A) File register, File $85h$
TRISB	TRIState B (Data Direction register B) File register, File $86h$
TRISC	TRIState C (Data Direction register C) File register, File $87h$
TRISD	TRIState D (Data Direction register D) File register, File $87h$
T0SE	Timer 0 Set Edge; OPTION_REG[4]
TTL	Transistor Transistor Logic family
TTY	TeleTYpewriter
TX	TranSmit pin for USART (Common with RC6)
TXEN	TranSmit register ENable; TXSTA[5]
TXIE	TranSmit register Interrupt Enable mask; PIE1[4]
TXIF	TranSmit register Interrupt Flag; PIR1[4]
TXREG	TranSmit data REGister; File $19h$
TXSTA	TranSmit STAtus register; File $98h$
μs	Microsecond (10^{-6} s)
UART	Universal Asynchronous Receiver Transmitter
USART	Universal Synchronous-Asynchronous Receiver Transmitter
V_{DD}	Positive (Drain) supply voltage
V_{EE}	Earth (0 V) supply voltage
V_{PP}	Positive Programming voltage
VLSI	Very Large-Scale Integration
W	Working register
WCOL	Write COLlision; SSPCON[7]
WR	WRite; EECON1[1]
WREN	WRite ENable; EECON1[2]
WRERR	WRite ERRor; EECON1[3]
Z	Zero flag in the Status register; STATUS[2]

Index