11

Computer Implementation

Having devised control algorithms and converted them to software of one form or another, our next step is to integrate the system to include a computer to run it on. For the experimental work, it has been easiest to exploit a retired PC, but for serious product development, some sort of computing engine must be integrated into the design as a whole.

There are some features that are common to the PC and to the humblest of microcontrollers, which can greatly influence your approach to the task.

11.1 ESSENTIALS OF COMPUTING

As the computer has evolved, many ingenious variations have been tried. Some have survived, while some have gone the way of the dodo. But some underlying principles remain unchanged.

11.1.1 General Fundamentals

The simplest computing engine is the *Turing machine*. This is really a figment of the mathematicians' imagination, used to decide what is "computable" or not. It has an input bit and a "state" signifying which "instruction card" is in play. From these, the output bit and the next instruction to be used are specified.

Essentials of Mechatronics, by John Billingsley

When we get to a "real" computer, the essentials are memory, program counter, and, of course, input and output. There are usually several sorts of memory. The most easily accessible are "registers" such as one or more accumulators to hold the value that any calculation has reached so far and RAM (random access memory), an array of "pigeonholes" in which numbers can be stored.

Any embedded system also has ROM (read-only memory) that contains code and data that cannot change. To complicate matters, there is also EAROM (electrically alterable ROM) to hold data that must survive the system being switched off.

We now have a program that is stored in memory. In Von Neumann machines, the great majority, this memory can double for both program and data, although some other devices have separate memory formats. To access a byte or word of memory, its *address* is placed on a *memory address bus.*

Most instructions will manipulate data, performing arithmetic or logic operations on values in the memory or registers and going on to execute the next instruction. The address of this instruction is held in the *program counter.* Other instructions will influence the program flow, with *branch* or *jump* instructions to allow a piece of code to be skipped or executed repeatedly. There are also *conditional jumps* to determine whether to jump according to the result of a comparison, so that a loop can be terminated after a number of executions or on the result of some input value.

Input–output is, of course, a vital operation without which the computer has no real purpose. The "classic" form of input is to transfer 8 or 16 bits, represented by logic voltages on an array of input connections, to an accumulator register within the processor. Input–output registers are often *memory-mapped*; in other words, they behave as though they are memory at some specific location, enabling values to be input and output as though reading from or writing to memory.

11.1.2 Subroutines

In the evolution of the computer, an important "bright idea" was the conditional jump. Another was the subroutine call. Suppose that we wish to perform a special operation on a number, such as evaluating its logarithm. We can write a block of code to perform the operation and include it in the software. Now suppose that we wish to evaluate the logarithm of another number, somewhere else in the program. We could, of course, plant a second copy of the logarithm code in the program, but this would waste space.

Instead we can "call" a single copy of the logarithm routine from a number of different parts of the program. This is different from a "jump," since we must know where to return afterward. We must also tell the routine the value that we wish to convert, and in turn the routine has to convey the answer when returning. This could be done by holding values in the registers, but the accepted method is to use the "stack."

Most processors have a *stack pointer*, holding a value that points to an area of memory which is otherwise uncommitted. A PUSH command will save a register's value in the address pointed to by the stack pointer, which will automatically increment (or decrement) to point to the next location. A POP command will do the opposite, reading from the stack (after decrementing or incrementing) and restoring the register. Various processors work in various ways, incrementing or decrementing before or afterward, but the programmer simply has to ensure that the PUSHes match the POPs.

Now, in a subroutine call, the program counter is pushed onto the stack, and its value is retrieved when the code "returns."

11.1.3 Interrupts

The next bright idea was the interrupt. Until then, the program execution had depended on the program itself, together with any values that are input and later used to influence conditional jumps. At various places in the software, a call might be made to a subroutine to check whether a new byte was ready for input. A loop that checks inputs in turn is termed a *polling loop*.

With an interrupt, the data-ready event grabs the attention of the computer and takes it immediately to the routine that will deal with it. The machinery pushes the return address onto the stack, together with the status register. The interrupt routine then starts to execute. Its first task is to save any register that might be changed in the routine, so that afterward the computer can pick up the action where it left off, just as though the interruption had not happened.

The interrupt can be caused by the arrival of data, by an external device being ready for another byte of output, by some sort of timer, or by an input event such as the pressing of an emergency button.

Despite their great advantages, interrupts are a nightmare for real-time troubleshooting. Except under the most artificial of conditions, the program will never be executed the same way twice. What is more, interrupts lead to a multitude of philosophical problems for the operating system designer. What happens if the computer is executing one device's interrupt routine when another interrupt arrives?

This leads to the idea of an interrupt hierarchy. An interrupt is permitted only if it is "more important" than any interrupt state existing.

But the philosophy gets deeper. When a data byte arrives on a high-speed serial connection, there is an urgent need to read it before the next byte arrives. The interrupt routine's purpose is clear. It must copy the byte to a "buffer" in memory, and then computing can resume.

What happens when the last byte is received and the data transfer is complete, however? How does the software decide which task should take precedence?

But first there is a faster means of dealing with data transfer: direct memory access (DMA).

11.1.4 Direct Memory Access

Some devices are capable of sending a burst of data. It is a wasteful operation to execute an interrupt for each byte, with the need to save registers and "environmental variables," then to input the byte and save it in the correct location, and then to restore the variables and return. Instead, DMA allows an external device to gain access to the memory address lines and plant the data in memory with no reference to the processor itself.

A "bus request" is pulled down, and when a "bus grant" is given, the transfer can begin. In the case of the PC, the term "external" is a relative matter. A DMA controller is built into the hardware and performs all the hard work. This is primed with the start address of the memory to be filled and the number of bytes or words to transfer. It clocks new bytes from the peripheral and saves them sequentially in memory. When the last has been received, it releases the bus request and, if desired, causes a hardware interrupt so that the data can be dealt with.

Of course, the process can work in reverse with the contents of a memory block being output.

11.2 SOFTWARE IMPLICATIONS

The ways of programmers are something of an enigma. On one hand, the GOTO statement is deprecated, for very sound reasons, while on the other hand flow diagrams are encouraged—yet every line in the flowchart is the embodiment of a GOTO statement!

In the early days of programming, overenthusiastic software writers were often guilty of "spaghetti code," with jumps in and out of loops that required great patience to trace. Another vice was the use of identifiers such as **`a`** or **`i5`**, which gave no clue as to their purpose or meaning.

But surely the pendulum has swung too far the other way, when identifiers such as

```
CoGetInterfaceAndReleaseStream
CoMarshalInterThreadInterfaceInStream
StgGetIFillLockBytesOnILockBytes
CoGetCurrentLogicalThreadId
WdtpInterfacePointer_UserMarshal
WdtpInterfacePointer_UserUnmarshal
```

are quite typical within a popular operating system.

Software tools allow great slabs of code to be stacked up in a pile that defies the efforts of the programmer to read through and check the fundamental details. But when real-time code is to be written for an embedded machine, there are great virtues in keeping it lean and mean. It is my opinion

that wherever possible, identifiers should be no longer than two syllables, whether they are variable names or procedures.

The choice of a computer language is not a simple matter. Language has many dimensions. First there is the "speak," the words and symbols that will be used to define the code. A page of Java will look very much like a page of C, while the line-by-line text of a Visual Basic program will look very much like other forms of Basic—and might even bring back distant memories of FORTRAN.

Underlying the code is the structure of its execution. What makes Visual Basic visual is its use of "forms" on which are placed "controls." Each control has a piece of code to deal with any "event," such as the click of a mouse, the operation of a "button," or the change in a "slider." This has every appearance of being real time, but in most cases the interrupts are illusory. There is an instruction, `DoEvents`, which really means "go and poll any other tasks that might need attention." A loop that does not include a `DoEvents` can lock up the machine so that user inputs are ignored.

QBasic or Quick Basic, on the other hand, will allow an event to grab control at the completion of any instruction. In general, the "lower" the level of a language, the more control the programmer will have over the way the code will be executed. C is close to assembly language and allows much greater control.

The "programming environment" is another important factor. A "user-friendly" system will check each line as it is entered and signal any syntax errors. In Visual Basic, a click on the RUN icon is all that is needed to test the code. When the program crashes, moving the cursor to any variable will cause its value to be shown in a TOOL TIP, while the offending line of code is highlighted in yellow. The programmer is faced only with the task of entering code that is "obviously necessary," plus the properties of controls, such as their background color, and the layout of the forms.

At present there are at least two "flavors" of C++ language. The Borland version leans toward Visual Basic, with controls that can be dropped into forms resulting in the automatic generation of the associated code. In the Microsoft version, there is much more housekeeping to do. First you must decide on what sort of project you require. Is it "bare screen," or do you wish to have forms and controls? Will it result in an exe file, a dll library, or a DirectX filter?

Then, after a "wizard" has set up the empty project files for you—although they might already look pretty crowded—you have to be concerned with both code files and "header" files that define how your functions are to be called.

Before you can run your project, you have to "build" it. Only then do you see a list of your errors. The omission of a single } brace can result in a list of a dozen or more errors. The "friendly features" are not made very clear in the documentation or help files, but after much exasperation you find that clicking on a line warning you of an error will actually take you to the offend-

ing line itself. By inserting breakpoints, you can achieve the same display of variable values that Visual Basic offered so easily.

So, why endure the hardships of riding bareback? C gives access to the "inner workings" of the machine in a way that is protected from VB users. Its closeness to machine code allows it to perform tasks that in VB would require the writing of special library routines—and these would probably be written in C in preference to assembler.

In C, you certainly have more control of the way the code is executed, but the promise of more efficient code than in Basic might be a false one. Consider the artificial and useless piece of Basic code:

```
DEFINT A-Z
DIM a(10),i
i = 5
a(i) = a(i) + i
```

The array and a variable are defined as integers, the value 5 is placed in *i*, and then the *i*th element of *a*( ) has *i* added to it. It looks as though the pointer into the array will have to be calculated twice and that a C version could be much more efficient:

```
void main(){
int a[10], i;
i=5;
a[i]+=i;
}
```

The cryptic final line of the C version certainly looks more compact. But Quick Basic has an optimizing compiler. When the resulting assembler code is listed, we see

```
mov   I%,0005h
mov   si,I%
sal   si,1
mov   ax,I%
add   A%[si],ax
```

which really could not be more efficient. In the last line, the value of *i* is added straight into the array, using an index that has been loaded with the correct value, then shifted left to make it a word pointer.

The C version is converted by Visual C++ to give

```
mov    WORD PTR _i$[ebp], 5
movsx  eax, WORD PTR _i$[ebp]
mov    cx, WORD PTR _a$[ebp+eax*2]
```

```
add   cx, WORD PTR _i$[ebp]
movsx edx, WORD PTR _i$[ebp]
mov   WORD PTR _a$[ebp+edx*2], cx
```

The addition is performed in a register that then has to be saved.

Writing code for a simple embedded processor is likely to have even fewer home comforts. It will involve first keying in the code as a text document. The assembler (or maybe a C compiler) is then invoked to convert the code and produce a binary "object file." This must then be downloaded to the processor in yet another operation. This must all be achieved before the code can be tested, and additional means must be devised for monitoring what the software is actually doing.

11.2.1 Structured Code

We have seen that a subroutine or "procedure" economizes on the space of code that might otherwise have to be repeated. It has another important role, however. It is a module of code that can be tested exhaustively and can then be called with a simple well-named command. A lengthy matrix inversion routine could appear in the program flow as just

```
invert a()
```

When the code is written at the assembler or C level, good structure is even more important. It is also essential to document the code with clear comments. The choice of identifiers can do much to improve readability, using verbs for procedures and nouns for variables.

I am greatly in favor of the use of "pseudocode," a language that exists in the mind of the program writer. Consider the task of writing code for a four-legged walking robot with vacuum grippers on its feet. A pace could involve moving each foot forward in turn and might be represented as follows:

```
Sub Pace()
   For foot = frontleft to hindright
      Lift foot
      Advance foot
      Place foot
   Next foot
End Sub
```

At this level, before getting bogged down in software details, it will be clear that taking a second pace will present the problem that all four feet are already in the forward position!

So it is no effort to make a note that when writing the Advance subroutine, while "this" foot is being moved forwards the other feet must be moved one-third of a stride to the rear.

In fact, this code example could become Visual Basic code as it stands. For other implementation, such as in assembly language or C, it could appear as the "remarks" that are added to make the code meaningful.

```
*      Sub Pace()
PACE   LDAA #FRONTLEFT * For foot = frontleft
       STAA FOOT       *     to hindright
PACE1  JSR  LIFT       * Lift foot
       JSR  ADVANCE    * Advance foot
       JSR  PLACE      * Place foot
       INC  FOOT       * Next foot
       LDAA FOOT
       CMPA #HINDRIGHT
       BLE  PACE1
       RTS             * End Sub
```

In this example, the pseudocode can burrow down a level to define

```
SUB lift(foot AS INTEGER)
   Unstick foot
   target(foot).z = target(foot).z+100
END SUB
```

in which we assume that there is an interrupt routine running in the background that handles position control.

The `Unstick` routine will output the signal that releases the vacuum, then will pause for an instant. The `target` line could instead output a value to an independent microcontroller dedicated to the control of that particular leg.

11.3 EMBEDDED PROCESSORS

While the microprocessors at the heart of a personal computer are still evolving in speed and complexity at an increasing rate, some of their humble cousins have remained in fashion for much longer. These are the simple processors that are embedded in washing machines, toys, clock radios, automobiles, and a host of other appliances.

A general electronics catalog has over 50 pages of microprocessors and microcontrollers, some of them from families virtually unchanged over 20 years. Of course, there have been numerous innovations, such as the ability to communicate over USB and CAN-bus, but assembly code written years ago can often still be adapted with relative ease.

The PC user may grumble at the length of time taken for the system to load, but will not consider the "boot" process as a personal worry. For the

designer of an embedded system, the entire startup process from the first application of power must be part of the design.

11.3.1 Essentials of a Microprocessor

The early 8-bit chips consisted of the processor alone, plus a number of internal registers. Random access memory (RAM) for calculations and read-only memory (ROM) to hold the program had to be wired on to address and data buses, usually with extra chips for address decoding plus a crystal to set the clock frequency.

Many of today's chips aim for an "all in one package" approach. They may have 256 bytes or more of RAM, sufficient for many embedded tasks, plus several kilobytes of program memory. For laboratory and development work, this can conveniently be EAROM, electrically alterable ROM. The contents of the EAROM can be changed by the processor itself, although this is often much slower than normal RAM operations. The data will remain unchanged when the processor is switched off and on again.

This is a comfortable size of program to handle in assembly language, but much larger memories are common. A "thumb drive" or MP3 player with less than 64 Mbyte of memory would be regarded as tiny.

Just as programming languages have their faithful adherants, enthusiasts will concentrate on a particular microcomputer system. My colleague, Mark Phythian, is especially supportive of the PIC computer. He has designed the simple application described below, in which it serves as an analog-to-digital converter, encoding and transmitting the results over a serial interface to the host PC.

By using this interface, many of the problems of the Windows operating system are bypassed. A Visual Basic program is outlined below that includes an MSComms component to handle the serial communications. The values of four channels are displayed on a form as a sort of "oscilloscope trace". From here it is a small step to using the chip for online control.

The chip is a PIC16F88 from Microchip. It has four 10-bit analog input channels, 4 bits of logic input, and 4 output bits. It also has two interrupt lines and a bidirectional serial interface. Two transistors with four resistors and a diode are needed to convert the serial logic levels to something compatible with the PC's RS232 interface, but with those the circuit is complete.

The circuit diagram in Figure 11.1 shows a potentiometer and pushbuttons that can be used to test the circuit's operation, but these are not part of the basic design.

When writing code from first principles, you can make up the command rules as you go along. But it is important that they be strictly structured in a well-defined protocol.

This particular software operates as follows. The PC sends a command as a single byte. The PIC responds with the same byte, followed by any data bytes that are requested.

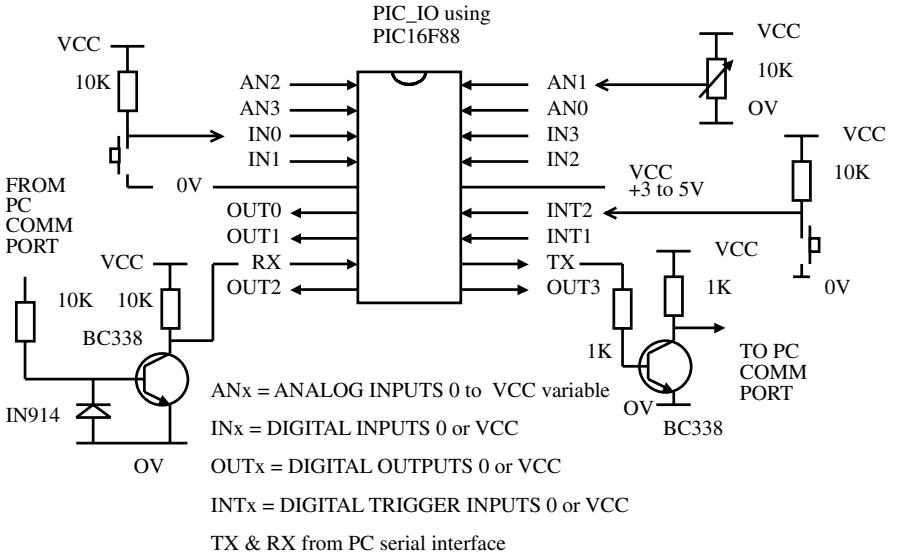

Figure 11.1 *Mark Phythian's circuit of single-chip microcomputer ADC.*

Hex $30 returns 8 bytes, representing four channels of 10-bit ADC readings.

Hex $40 returns 4 bytes, representing four channels of 8-bit ADC readings.

Hex $50 returns 1 byte, with the lower 4 bits representing the input bits.

Hex $60 to $6F will cause the lower 4 bits to be sent to the output pins.

For the two interrupt pins on the PIC (active low):

INT1 sends 1 byte hex $21 character.

INT2 sends 1 byte hex $22 character.

Even with such a simple chip, it is not necessary to revert to assembly language. A Basic *cross-compiler* can be purchased for about thirty Australian dollars from http://www.oshonsoft.com/, and it is for this system that Mark has written his code.

Even so, it is necessary to attend to every detail of setting up the chip's state, defining variables and enabling the necessary interrupts to handle communications:

```
' PIC_IO_BAS.bas PIC serial IO interface by Mark Phythian
' Uses Microchip PIC16F88 processor running at 8MHz
internal RC Osc.

' define variables
```

```
Dim adtable(4) As Word ' table to hold adc results
Dim chan As Byte       ' channel number
Dim val As Word        ' word size adc result variable
Dim oldb As Byte       ' last portb value
Dim char As Byte       ' single character command
Dim command As Byte    ' upper 4 bits of command
Dim n As Byte          ' lower 4 bits of command
Dim m As Byte          ' byte size temporary variable
Dim wrd As Word        ' word size temporary variable

' setup PIC
OSCCON = 0x72          ' set internal RC select to 8MHz
TRISB = %11000000      ' set PORTB 0-5 pins as outputs,
                       '6 & 7 inputs
Gosub initad           ' initialise adc
PIE1.RCIE = 1          ' enable UART RX interrupt
OPTION_REG = 0x7f      ' enable PORTB weak pullups for
                       inputs 6 & 7
Hseropen 57600         ' set UART BAUD rate
INTCON = 0xc8          ' enable GIE, PEIE and RBIE

' initialise variables
PORTB = 0x00
oldb = PORTB And 0xc0  ' last value of PORTB bits 6 &
                       '7 for change of state
WaitMs 1000
Hserout "OK"           ' startup ok
WaitMs 100

' endless loop converting as fast as possible
main:

ADCON0 = 0xc1             ' select channel 0 in bits
                          '5,4,3 of ADCON0

For chan = 0 To 3
  Gosub adconv       ' go to conversion routine
  val.HB = ADRESH    ' save high byte (upper 2 bits
                     'only)
  val.LB = ADRESL    ' save low byte
  adtable(chan) = val
  ADCON0 = ADCON0 + 0x08 ' increment selected channel
Next chan
Goto main 'repeat forever
End
```

```
' Initialise ADC
initad:
TRISA = %11111111       ' set portA as input
ANSEL = %00001111       ' set PORTA pins 0-3 as analog
                        'inputs
ADCON1 = 0x80           ' set 10 bit A/D result format
                        ' right justify ADRESH/L
ADCON0 = 0xc1           ' set A/D conversion clock to
                        'internal source,
                        ' turn on adc
Return

' Adc conversion routine
adconv:
High ADCON0.GO_DONE     ' start the conversion
While ADCON0.GO_DONE    ' wait until conversion is
                        'completed
Wend
Return

On Interrupt
Save System
' check for PORTB change of state on bits 6 & 7
If INTCON.RBIF = 1 Then ' test if portb change flag is on
  n = PORTB And 0xc0
  INTCON.RBIF = 0       ' reset RBI flag
  n = oldb Xor n
  If n.7 = 1 Then ' if bit 7 changed
    If oldb.7 = 1 Then ' if bit 7 changed to 0
      Hserout 0x22      ' send a " for INT2 input trigger
    Endif
  Else
    If n.6 = 1 Then    ' if bit 6 changed
      If oldb.6 = 1 Then ' if bit 6 7 changed to 0
      Hserout 0x21   ' send a ! for INT1 input trigger
      Endif
    Endif
  Endif
  oldb = PORTB And 0xc0 ' set oldb to new PORTB value
Else

' test for serial command received
  If PIR1.RCIF = 1 Then ' test if RXer flag is on
    PIR1.RCIF = 0       ' reset RCI flag
    Hserget char        ' get the received character
```

```
    command = char And 0xf0 ' command is upper 4 bits
    n = char And 0x0f   ' number is lower 4 bits

' fetch 10 bit adc values, returns 2 bytes each,
' command letter 0 (zero)
    If command = 0x30 Then
      Hserout char     ' echo command
      For n = 0 To 3
         Hserout adtable(n)  ' send 10 bit value in 2
                             'bytes
      Next n
    Else

' fetch 8 bit adc values, returns 1 byte each, command
letter @
      If command = 0x40 Then
        Hserout char         ' echo command
        For n = 0 To 3       ' D command requests all 4
          wrd = adtable(n)
          m = ShiftRight(wrd, 2)
          Hserout m          ' send 8 bit value as 1 byte
        Next n
      Else

' read inputs bits PA4-7, returns 1 byte, command
'letter P
        If command = 0x50 Then
          Hserout char ' echo command
          n = PORTA And 0xf0
          n = ShiftRight(n, 4)
          Hserout n          ' send 4 bits in low part
                             'of byte
        Else

' set outputs, command letters (from $60-$6F)
',a,b,c,d . . . o
          If command = 0x60 Then
            Hserout char ' echo command
            m = n And 0x03 ' arrange bits to Port B
            bits 4,3,1,0
            n = n And 0x0c
            n = ShiftLeft(n, 1)
            PORTB = m Or n ' set output bits from
            number n
          Endif
```

```
        Endif
      Endif
    Endif
  Endif
Endif
exit:
Resume
```

This code and the code for the Visual Basic test program can be found at www.essmech.com/11/3/1.htm.

The Visual Basic form has an MSComms control named `Serial` and a button with the name and caption `Quit`. Its code is as follows:

```
Dim bits10 As Byte          'For holding command
                            definitions
Dim bits8 As Byte
Dim getpins As Byte
Dim setpins As Byte
Dim bytes_in() As Byte
Dim Adc(3) As Single        'To hold ADC values between
                            '-1 and 1
Dim stopped As Boolean

Private Sub Form_Load()     'Execution starts here
Dim i As Integer
Show
Serial.Settings = "57600,n,8,1"          'make sure same
                                         'baud as PIC
Serial.CommPort = 1
Serial.Handshaking = comNone             'no handshake
Serial.InputMode = comInputModeBinary    'not ASCII text
Serial.NullDiscard = False               'treat nulls as
                                         'valid characters
Serial.PortOpen = True                   'open port

bits10 = &H30   'encode 4 channels, return 8 bytes of
                '10-bit data
bits8 = &H40    'encode 4 channels, return 4 bytes of
                '8-bit data
getpins = &H50  'read input pins, return in lower four
                'bits
setpins = &H60  'add required bit values to the lower 4
                'bits

Print "Cannot find PIC"   'Write warning message
Command setpins           'This will hang if PIC is not
                          present
```

```
Scale (0, 1)-(1000, -1)
Cls                        'Erase the message if all OK
stopped = False
Do Until stopped
   For i = 1 To 1000
      Adc10                          'contains DoEvents
      PSet (i, Adc(0)), vbBlack 'Plot the ADC values
      PSet (i, Adc(1)), vbRed
      PSet (i, Adc(2)), vbBlue
      PSet (i, Adc(3)), vbGreen
   Next
   Cls                               'Clear at end of trace
Loop
End                                  'End if the loop exits
End Sub

Sub Command(a As Byte)  'will flush buffer if necessary,
                        'hang if no PIC
Dim b(0) As Byte
b(0) = a
send b()
Do                                   'Wait for the echo byte
   get_bytes 1
Loop Until bytes_in(0) = a
End Sub

Private Sub Quit_Click()
stopped = True
End Sub

Sub Adc10()                          'Get four ten-bit values
Dim i As Integer
Command bits10
For i = 0 To 3
   get_bytes 2               'next line scales to range
                             '-1 to 1
   Adc(i) = (256! * (bytes_in(1) And 3) + bytes_in(0)) /
   512! - 1
Next
End Sub

Sub Adc8()                           'Get four eight bit
                                     'values
Dim i As Integer
Command bits8
For i = 0 To 3
   get_bytes 1
```

```
   Adc(i) = bytes_in(0) / 128! - 1  'scale
Next
End Sub

Sub get_bytes(n As Integer)    'Read from serial port to
                               'bytes_in()
    buf n                      'wait until n bytes
                               'received
    Serial.InputLen = n
    bytes_in() = Serial.Input
End Sub

Sub buf(i As Integer)          'waits for buffer to hold
                               'i bytes
    Dim j As Integer
    Do
       DoEvents
       j = Serial.InBufferCount
    Loop Until j >= i
End Sub

Sub send(a() As Byte)
    Serial.Output = a()
End Sub
```

The alternative to using a language such as Basic or C for the PIC code is to use assembly language. Mark Phythian has provided a sample of the equivalent code for this example, with just a small portion of the code involved:

```
; define variables
adtable EQU 0x39    ; adc result table 8 bytes
chan  EQU 0x41      ; channel no
val   EQU 0x42      ; word size adc result variable
oldb  EQU 0x44      ; last portb value
char  EQU 0x45      ; single character command
command EQU 0x46    ; upper 4 bits of command
n     EQU 0x47      ; lower 4 bits of command
m     EQU 0x48      ; byte size temporary variable
wrd   EQU 0x49      ; word size temporary variable

; Code executes here at start up
      ORG 0x0000    ;Location to put the code
      BCF PCLATH,3
      BCF PCLATH,4
      GOTO start
```

```
      ORG 0x0004    ;Place interrupt code here at
      address 0004
      MOVWF W_TEMP  ; save registers
      SWAPF STATUS,W
      CLRF STATUS
      MOVWF STATUS_TEMP
      CALL ISR           ; call interrupt service
                           routine
      SWAPF STATUS_TEMP,W
      MOVWF STATUS
      SWAPF W_TEMP,F
      SWAPF W_TEMP,W     ; restore registers
      RETFIE             ;return from interrupt

start:
; setup PIC
      BSF STATUS,RP0     ; select page 1
      MOVLW 0x72
      MOVWF 0x0F     ; set internal RC select to 8MHz
      MOVLW 0xC0
      MOVWF 0x06     ; set PORTB 0-5 pins as outputs,
                     ;6 & 7 inputs

; initialise adc
      MOVLW 0xFF
      MOVWF 0x05     ; set portA as input
      MOVLW 0x0F
      MOVWF 0x1B     ; set PORTA pins 0-3 as analog
                     ;inputs
      MOVLW 0x80
      MOVWF 0x1F     ; set 10 bit A/D result format
                     ;right justify ADRESH/L
      BCF STATUS,RP0     ; select page 0
      MOVLW 0xC1
      MOVWF 0x1F     ; set A/D conversion clock to
                     ;internal source,
                     ; turn on adc
      BSF STATUS,RP0     ; select page 1
      MOVLW 0x7F
      MOVWF 0x01     ; enable PORTB weak pullups for
                     ;inputs 6 & 7

; setup UART
      BSF STATUS,RP0     ; select page 1
      BSF 0x0C,5     ; enable UART RX interrupt
```

```
      MOVLW 0x08
      MOVWF SPBRG          ; set UART BAUD rate 57600
      BSF TRISB,2
      BSF TRISB,5          ; set PORTB bits 2 and 5 as
                           ;outputs for UART
      MOVLW 0x24
      MOVWF TXSTA          ; enable Transmitter
                           ; set High BAUD rate select bit
      BCF STATUS,RP0
      MOVLW 0x90
      MOVWF RCSTA          ; enable Serial port,
                           ; continuous enable receiver
       MOVLW 0xC8
       MOVWF 0x0B      ; enable GIE, PEIE and RBIE for
                       ;UART

; initialise variables
      BCF STATUS,RP0       ; select page 0
      CLRF 0x06        ; clear PORTB
      MOVLW 0xC0
      ANDWF 0x06,W
      MOVWF oldb       ; last value of PORTB bits 6 & 7
                       ;for change of state
; Send "OK"            ; start up ok
      MOVLW "O"
      CALL TXD
      MOVLW "K"
      CALL TXD
```

This does not yet include the receipt and execution of commands. It is clear that the use of the Basic compiler saves a large amount of effort.