Controlling the

from your Armchair.

Sony, Infrared Remote Control Decoding.

Infrared remote control decoding is considered something of a black art, however, this tutorial will show you that its principals are quite straightforward, and easy to implement on a PIC microcontroller.

Infrared remote control has been around for a very long time now, and we tend to take it for granted. Yet it's a marvel of modern technology, which allows a whole variety of devices to be activated with the touch of a button. Remote control handsets are so abundant that they may be purchased new for a few pounds, which makes them viable items for experimentation. There are dedicated chips available that will decode the signals from a particular handset, however, with the flexibility and cost effectiveness of the PIC range of microcontrollers we can develop a decoding subroutine that may be placed into your own programs, or used as a stand-alone infrared to RS232 converter.

Manufacturers Protocols.
Regrettably, remotes do not come in a single flavour, each manufacturer uses a different set of protocols. The three main ones are RC80, which is used by Panasonic. RC5, which was designed by Philips and is one of the more popular types, and then there's the Sony protocol, named S.I.R.C, which is hugely popular and also one of the simplest to decode. Therefore, I will take the less complex type and endeavour to illustrate how to decode the signals from a Sony remote control handset, using the ever-popular PIC16F84 microcontroller.

Infrared to TTL Converter.
In a bid to eliminate ambient light sources, both natural and manmade, from interfering with the data stream transmitted by the handset, modulated light is used. This modulation is centred around different frequencies depending on the manufacturer; and varies from 32KHz to 40KHz. In the case of Sony handsets, the modulation is centred at 40KHz, which means we require a device that can receive the modulated infrared light and convert it into a TTL signal that the PIC can handle.

There are a number of these devices available, each having a specific centre frequency that they're more sensitive too. The device used for this tutorial is the IS1U60 from Sharp. It has a centre frequency of 38KHz, which is close enough to 40KHz so as not to matter. Figure 1, shows the internal block diagram of one of these devices.

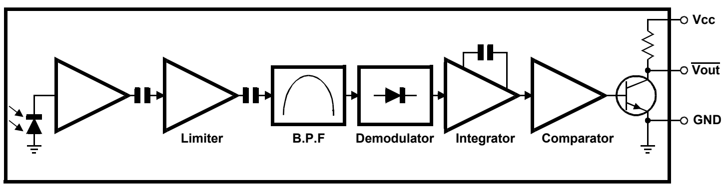

Fig 1. Internals of the IS1U60.

As you can see, these deceptively simple looking devices are a lot more than a re-packaged IR Photodiode. They filter, amplify and demodulate the infrared signal. Then give a nice clean TTL output by means of a final comparator stage. They also have a built in automatic gain control (*AGC*), which helps stop overloading if the handset is held too close. Using one of these devices is a great deal cheaper (and easier) than building your own discrete version.

Most IR sensors have an active low output, which means that the PIC is presented with a logic 0 when an infrared signal is detected. With no signal present, a maximum current of 4.8mA is consumed (2.8mA being typical). In addition, the recommended voltage is 4.7V to 5.3V.

Sony Protocol (S.I.R.C).

S.I.R.C (Serial Infra-Red Control) uses a form of pulse width modulation (*PWM*) to build up a 12-bit serial interface, known as a *packet*. This is the most common protocol, but 15 bit and 20-bit versions are also available. A pulse with a duration of 2.4ms is sent first as a header, this allows the internal AGC to adjust and also allows the receiver to check if a valid packet is being received. A 1-bit is represented by a pulse duration of 1.2ms, while a 0-bit has a duration of 0.6ms. A delay of 0.6ms is placed between every pulse.

The string of pulses build up the 12-bit packet consisting of a 5-bit (0..31) device code, which represents a TV, Video, Hi-Fi etc (see table 1), and a 7-bit (0..127) button code, which represents the actual button pressed on the remote (see table 2). The packet is transmitted most significant bit first (*MSB*), with the device code being sent, then the button code. Figure 2, illustrates this more clearly. After the packet is sent, a delay is implemented, which brings the whole transmitted signal to a length of 45ms. This is repeated for as long as a button is pressed.

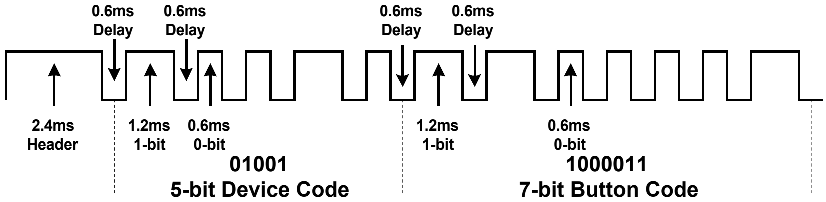

Fig 2. 12-bit packet construction

Command	Device
1	Television
2	VCR 1
4	VCR 2
6	Laser disk player
12	Surround sound unit
16	Cassette deck/tuner
17	CD player
18	Equaliser

Table 1. SIRC device code.

Command	Function
0-9	Numerals 0 to 9
16	Channel +
17	Channel -
18	Volume +
19	Volume -
20	Mute
21	Power
22	Reset
23	Audio mode
24	Contrast +
25	Contrast -
26	Colour +
27	Colour -
30	Brightness +
31	Brightness -
38	Balance left
39	Balance right
47	Power off

Table 2. SIRC TV button code.

Assembler vs. BASIC.

Knowing the principals behind infrared communications is one thing, actually writing software based on the information is a whole new ball game. Whenever PICmicros are mentioned, people tend to think of the rather cryptic language of assembler, however, this is not the case anymore, as there are many high level language implementations for use with the PICmicro, such as C, C++, Pascal, and BASIC. My personal preference is BASIC. The BASIC language in general has received a lot of bad press since its conception in the middle part of the 70's and is considered to be clumsy and inflexible, yet nothing could be further from the truth. Thanks to the PICBASIC PLUS and melab's PICBASIC compiler range, this language has been brought into the 21st century. Thanks also, in part, to BASIC's shallow learning curve, software designs that used to take weeks can now be realised in a just few hours.

So as to not seem too biased towards either language, I will present the software for this article in both assembler and PICBASIC PLUS, which will enable you to choose your preferred type. I will also endeavour to illustrate the pro's and cons of both languages by not using optimised assembler routines. This means that both the BASIC and the assembler versions will follow the same structuring, which will enable a fairer appraisal of them. It is not my intention to teach you how to program a PICmicro, therefore, throughout this article, it is assumed that you already have some knowledge of either assembler or PICBASIC PLUS. And that you have a means of programming the PIC16F84.

For more information concerning the PICBASIC PLUS compiler, as well as an assortment of programmers and information regarding the PICmicro in general, visit Crownhill Associate's dedicated web site at **www.crownhill.co.uk**. For information concerning assembler programming, visit Microchip's web site at **www.microchip.com**.

Circuit Description.
In order to demonstrate the principals behind infrared decoding, the circuit in figure 3 is employed. The PIC circuit incorporates two light emitting diodes, one green and the other, red. The software is arranged in such a way that by pressing the channel-up button on a TV remote, the green LED will illuminate and channel-down will illuminate the red LED. As well as illuminating the LED's, two bytes are transmitted serially (*Async RS232*) from bit-3 of PORTA through a 1kΩ current limiting resistor (R2). The serial data contains the device code as well as the button code and is transmitted at inverted 9600 baud (N-8-1). Possible uses for this could be to attach it to the PC's serial input for remotely controlling some software, or for use in a robotics construction. The circuit layout is not too critical and could easily be built on a piece of stripboard. However, decoupling capacitor C5 should be placed as close to the IR sensor as possible, and C2 should also be located close to the PIC.

Figure 4 shows a possible layout for the circuit on a solderless breadboard.

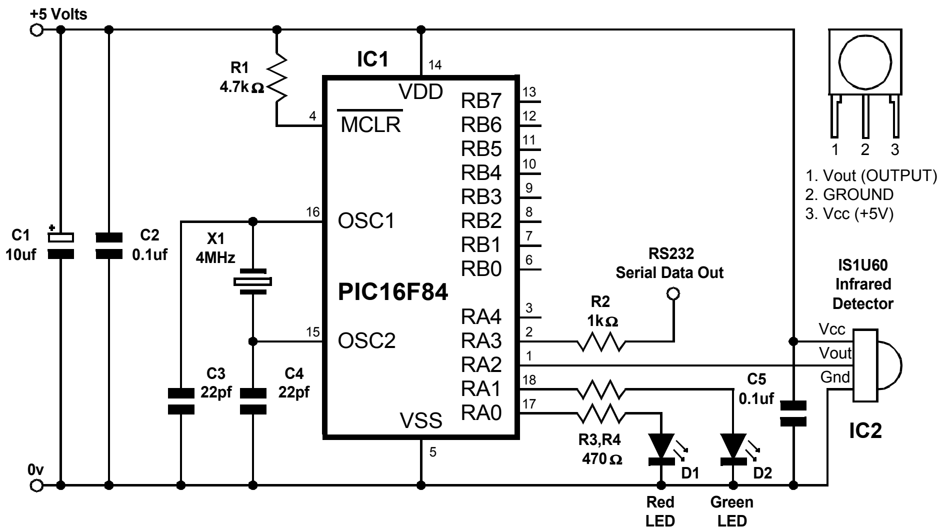

Fig 3. Sony, Infrared Remote control decoder circuit.

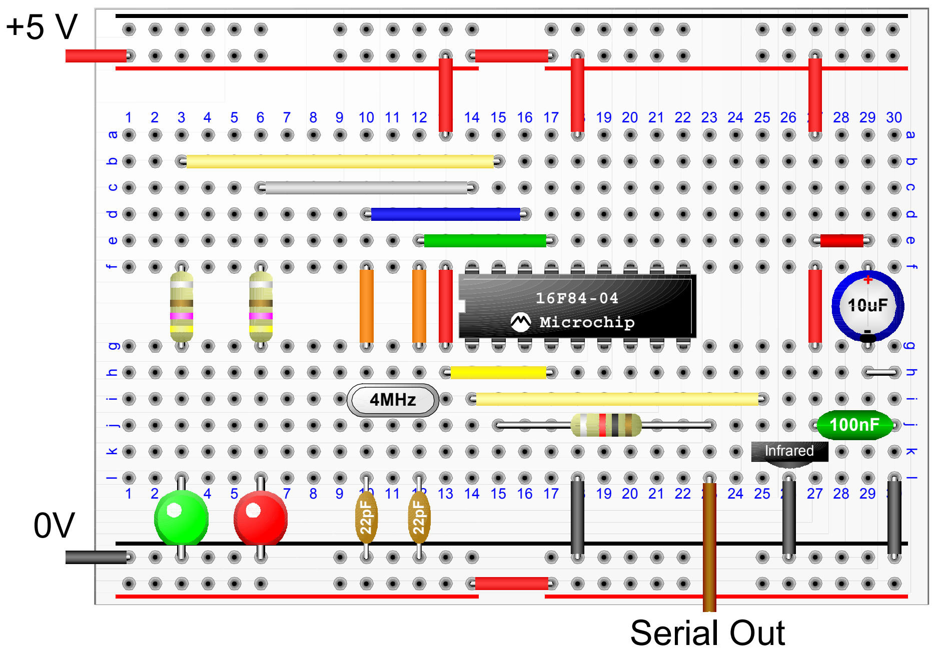

Fig 4. Possible solderless breadboard layout of above circuit..

Getting down to the coding.

The actual infrared decoding software is presented in the form of a subroutine, named **IRIN**, which will ease the inclusion of it into your own programs. The subroutine and subsequent main program loop may be split into several software tasks. These are outlined below: -

- **Task 1**..........Configure PORTA for both Inputs and Outputs
- **Task 2**..........Devise a method of measuring the high to low pulse length received from the active low IR sensor.
- **Task 3**..........Implement task 2 to detect the header and bit pulses and then construct the 12-bit packet.
- **Task 4**..........Split the packet into two separate bytes containing the 7-bit button and 5-bit device codes.
- **Task 5**..........Devise a method of transmitting inverted serial RS232 data.
- **Task 6**..........Construct a main program loop that calls the decoder subroutine and illuminates the correct LED, as well as using task 5 for transmitting both, the device and button codes, serially.

Task 1.

Our first coding task, that of configuring the Port's direction, is the easiest to accomplish. The assembler code for this is shown in listing 1. This will configure bits 0, 1 and 3 of PortA as outputs, for the attachment of the LEDs as well as the serial output. Bit-2 is made an input for the attachment of the IR sensor.

```
Bsf     STATUS,5 ;Point to TRIS reg
Movlw   00000100b ;Set PORTA,2 as IN
Movwf   PORTA     ;Configure the Port
Bcf     STATUS,5 ;Back to Page0
```

Listing 1. Assembler, Port direction.

The same thing written in PICBASIC PLUS is: -

```
TRISA = %00000100
```

Note, that there is no actual need to do this in PICBASIC, as the commands that deal with external influences automatically set the required pins as inputs or outputs.

Task 2.

Our second coding task, is a means of measuring the pulse durations that signify a header, as well as the separate ones and zeros that go to make up the packet. An assembler version of a routine that will do just this is shown in listing 2. The high to low pulse duration is measured at bit-2 of PortA and the 8-bit value is returned in the variable **P_VAL**. Because we're using an 8-bit (*0..255*) variable, it's impossible to return a value of 2400 for a pulse length of 2400 microseconds. Therefore, the routine has a resolution of approx 11microseconds when used in conjunction with a 4MHz crystal. An 11us resolution was chosen as opposed to 10us, because not all remote handsets stick stringently to the recommended pulse widths. Therefore, a header pulse could be more than 2.55ms in length, which would push it beyond a byte's storage capacity, i.e. greater than 255.

```
; Measure the duration of a high to low pulse on PORTA,2
; And leave the result in P_VAL.
; An 11us resolution is achieved with a 4MHz crystal.
Pulsin  Clrwdt           ; Walk the dog
        Clrf    Cntr     ; Clear the variables used,
        Clrf    P_Val    ; prior to the subroutine
Trans   Btfss   PORTA,2  ; Wait for a 1-to-0 transition
        Goto    Edge     ; Edge found!
        Incfsz  P_Val    ; Else increment P_VAL until >255
        Goto    Trans
        Incfsz  Cntr     ; Loop until 255
        Goto    Trans
        Return
Edge    Clrf    P_Val    ; A 1-to-0 transition occurred
Ege_lp  Btfsc   PORTA,2  ; Count how long it's logic 0
        Return
        Clrwdt           ; Walk the dog
        Nop              ; Timing loop
        Nop
        Nop
        Nop
        Nop
        Incfsz  P_Val    ; Increment P_VAL until > 255
        Goto    Edge_LP
        Return
```

Listing 2. Assembler, pulse measurement subroutine.

The values returned in **P_VAL** for a given pulse length are as follows: -
Header pulse… 2400us will return 220.
One-bit pulse… 1200us will return 110.
Zero-bit pulse… 600us will return 55.

To do the same task in PICBASIC PLUS, requires just one command: -

```
VARIABLE = PULSIN PORTA.2 , LOW
```

When used in association with a 4MHz crystal, the compiler's **PULSIN** command has a resolution of 10 microseconds. Also, if a 16-bit variable is used to hold the result then a duration of 0.. 65535us may be measured, where as, if an 8-bit variable is used, this is reduced to 0..255us. We can use this property to our advantage by detecting the 2400us header pulse with a 16-bit variable, and the individual 600us or 1200us bit pulses with an 8-bit variable. This will eliminate any problems arising from a header pulse that is longer than 2.55ms.

The values returned from the **PULSIN** command are as follows: -

Header pulse…2400us will return 240.
One-bit pulse… 1200us will return 120.
Zero-bit pulse… 600us will return 60.

The end parameter of the **PULSIN** command HIGH or LOW (*1 or 0*), determines whether a high-to-low pulse, or a low-to-high pulse is to be measured. Where a LOW or zero, measures a high-to-low pulse.

Task 3.

We now come to one of the two main body parts that build up the subroutine **IRIN**, in which we gather the bit information received from the IR sensor and construct the 12-bit packet.

The assembler version of this is shown in listing 3. The first thing the routine does is to try and detect a 2.4ms header pulse using the **PULSIN** subroutine (*task 1*). The result, held in **P_VAL** is examined to see whether it's between the values of 200 and 250. If it does not lie between these values, then the subroutine is exited with **IR_DEV** and **IR_BUT** holding a value of 255, which signifies an invalid header. If, however, a valid header IS detected, then a loop of 12 is set up. Within this loop, the individual bits are measured using the subroutine, **PULSIN**. Depending on the result returned in **P_VAL**, the individual bits of the 16-bit variable **PACKET** are set or cleared. This is achieved by splitting the difference between a one-bit (*110*), and a zero bit (*55*). If the result is greater than or equal to 80 then it must be a one-bit that's been received and if it's less than 80 then it must be a zero-bit.

```
; Receive a signal from a Sony remote control
; If no header then IR_DEV, IR_BUT will hold 255
IRIN     Clrwdt             ; Walk the dog
         Call     Pulsin    ; Measure the header
; Verify a good header, if its not valid then exit
; ** If PVAL < 200 then return with IR_DEV=255 **
         Movlw    200
         Subwf    P_VAL,W
         Btfsc    STATUS,C
         Goto     Next1
         Movlw    255
         Movwf    IR_Dev
         Return
; ** If PVAL > 250 then return with IR_DEV=255 **
Next1    Movlw    250
         Subwf    P_VAL,W
         Btfss    STATUS,C
         Goto     PK_Strt
         Movlw    255
         Movwf    IR_Dev
         Return
; Build up the packet, by pulling in all 12 bits
PK_Strt Movlw     12        ; Create a loop for 12 bits
         Movwf    Bitcnt
S_again Call      Pulsin    ; Get the bit duration
         Movlw    80        ; If it's >= 80 then it's a 1
         Subwf    P_VAL,W
         Btfsc    STATUS,C
         Goto     One
         Bcf      Packet+1,4 ; Clear the bit
         Goto     Cont
One      Bsf      Packet+1,4 ; Set the bit
Cont     Rrf      Packet+1,F ; Rotate bit into place
         Rrf      Packet,F
         Decfsz   Bitcnt    ; Have we done 12 bits yet?
         Goto     S_again   ; No! then loop again
```

Listing 3. Assembler, 12-bit Packet constuction.

The PICBASIC PLUS version of the same routine is shown in listing 4. This has exactly the same function as the assembler version, however, because of the different values returned from the **PULSIN** command, the comparisons for a header and bit pulses are slightly different. The resulting 12-bit packet for both types of routine are held in the variable **PACKET,** ready for splitting into its separate codes.

```
' Receive a signal from a Sony remote control, return with the 7-bit
' BUTTON code in the variable IR_BUT and the 5-bit DEVICE code in the
' variable IR_DEV If no header detected then IR_DEV, IR_BUT will hold 255
IRIN:   IR_Dev = 255
        IR_But = 255                   ' Preset the return variables
        Header = PULSIN IR_Sensor,LOW  ' Measure the header length.
        IF Header < 200 then RETURN    ' Verify a good header
        IF Header > 270 then RETURN    ' If not valid then exit
' Receive the 12 data bits and convert them into a packet
        Sony_Lp = 0
        REPEAT                 ' Implement a loop for the 12 bits (0 - 11)
        Packet.11 = 0          ' Default to a clear bit (zero-bit)
        P_Val = PULSIN IR_Sensor,LOW  ' Measure the LOW pulse width
        IF P_Val >= 90 then Packet.11 = 1 ' If pulse >= 90 then we've
                                       ' received a 1
        Packet = Packet >> 1           ' Shift the bits right 1 place
        INC Sony_Lp                    ' Increment the loop counter
        UNTIL Sony_Lp = 11             ' Close the loop after 12 bits
```

Listing 4. PICBASIC PLUS, 12-bit Packet constuction.

Task 4.

For the resulting 12-bit packet to be of any practical use, it must be split into the 5-bit device code and the 7-bit button code. This is achieved by a series of rotations then masking. The assembler version of this is shown in listing 5. Within the variable **PACKET**, the button code is located, starting at bit-0. This is now extracted by ANDing **PACKET** with 127 (*01111111*) and the result is placed into **IR_BUT**. To extract the device code, seven right rotations are performed, which will effectively move the button code out of the way and place the device code starting at bit-0 of **PACKET**. Again, this is extracted by ANDing, but this time with 31 (*00011111*) and placed into **IR_DEV**. The PICBASIC PLUS version of the same routine takes only two lines of code: -

```
; Split the 7-bit BUTTON code, and the 5-bit DEVICE code
        Movf    Packet,W ; Mask the 7-bit BUTTON code
        Andlw   01111111b
        Movwf   IR_But
; ** Shift PACKET and PACKET+1, right, 7 times **
        Rrf     Packet + 1,F
        Rrf     Packet,F
        Rrf     Packet + 1,F
        Rrf     Packet,F
        Rrf     Packet + 1,F
        Rrf     Packet,F
        Rrf     Packet + 1,F
        Rrf     Packet,F
        Rrf     Packet + 1,F
        Rrf     Packet,F
        Rrf     Packet + 1,F
        Rrf     Packet,F
        Rrf     Packet + 1,F
        Rrf     Packet,F
        Movf    Packet,W ; Mask the 5-bit DEVICE code
        Andlw   00011111b
        Movwf   IR_Dev
        Return
```

Listing 5. Assembler, Device code splitter.

```
' Split the 7-bit BUTTON code and the 5-bit DEVICE code
IR_But = Packet & %01111111        'Mask the 7 BUTTON bits
IR_Dev = %00011111 & (Packet >> 7) 'Move down and mask the 5 DEVICE bits
```

Task 5.

Our finished decoder could simply bring the eight PORTB pins high for a given button pressed on the handset, but a more elegant, and possibly more desirable result would be to transmit both the button and the device codes serially. Therefore, our fifth task is a subroutine that does just that. Listing 7 shows the assembler version of an async RS232 transmitter, operating at inverted 9600 baud from bit-3 of PORTA. The byte to transmit is first loaded into the W register then a call is made to **SOUT**. As it stands, the baud rate is set at 9600, however, to change it, alter the value placed into **DLCTR**, the higher the value, the longer the delay, thus, the lower the baud rate. For example, a value of 44 will lower it to 4800 baud, while 88 will produce 1200 baud.

To do the same task in PICBASIC PLUS, again takes only one command: -

```
SEROUT PORTA.3 , N9600 , [ Variable { , or variables } ]
```

PICBASIC PLUS's various serial out commands have a lot more tricks up their sleeves. Not only do they allow different baud rates from 300 to 38400; both inverted and non-inverted, but also output the results as 8 or 16-bit decimal, hexadecimal, binary or ASCII strings. This is ideal for interfacing to the many serially controlled LCD modules on the market.

```
; Transmit the byte held in W at inverted 9600 baud (8-N-1)
; from bit3 of PORTA.
Sout    Movwf   Tr_Byte ; Load TR_BYTE with W reg
        Movlw   8
        Movwf   Bit_Cntr ; Create a loop of 8
        Bsf     PORTA,3 ; Send the start bit
        Call    Bit_Dly ; Delay one bit time
Xmtlp   Rrf     Tr_Byte ; Rotate Right, moves data bits
                        ; into Carry, starting with bit-0.
        Btfsc   STATUS,C ; Is it a One-bit?
        Bcf     PORTA,3 ; Yes, so send A One
        Btfss   STATUS,C ; Is it a Zero-bit?
        Bsf     PORTA,3 ; Yes, so send A Zero
        Call    Bit_Dly ; Delay one bit time
        Decfsz  Bit_Cntr ; Have we reached 8-bits yet?
        Goto    Xmtlp   ; No, so loop again
        Bcf     PORTA,3 ; Yes, so send the stop bit
        Call    Bit_Dly ; Delay one bit time
        Return
; ** Delay 1-bit time subroutine**
Bit_Dly Movlw   22      ; Set Baud to 9600
        Movwf   Dlctr
Slp     Clrwdt          ; Walk the dog (1us)
        Decfsz  Dlctr
        Goto    Slp
        Return
```

Listing 7. Assembler, Serial output subroutine.

Task 6.

Our final task is to write the main program loop which will; call the decoder subroutine, serially transmit both codes, and illuminate the correct LED for a chosen button pressed on the handset.

An assembler version of this is shown in listing 8. Within the loop, the returning values from **IRIN** are examined, if **IR_DEV** returns holding 255 then an invalid header was detected so the process is repeated. If a valid header WAS detected, then both **IR_DEV** and **IR_BUT** are transmitted using the **SOUT** subroutine. A check is than made of **IR_DEV**, if it's not holding a value of one, then it is not a television remote handset, and again, the process is repeated. If however, the device code is for a television, **IR_BUT** is examined, if it holds a value of 16 (*channel-up*) then the green LED is turned on, and the red LED is turned on if it's holding 17 (*channel-down*).

```
; ** THE MAIN PROGRAM LOOP STARTS HERE **
Again   Clrwdt          ; Walk the dog
        Call    IRIN    ; Get the IR signal from the handset
        Bcf     PORTA,0 ; Turn off both LEDs
        Bcf     PORTA,1
        Movlw   255     ; If IR_DEV=255 then look again
        Subwf   IR_Dev,W
        Btfsc   STATUS,Z
        Goto    Again
; ** Transmit the DEVICE code then the BUTTON code serially
; ** at inverted 9600 baud N-8-1 **
        Movf    IR_Dev,W
        Call    Sout
        Movf    IR_But,W
        Call    Sout
; ** If IR_DEV<>1 (TV device code) then look Again **
        Movlw   0
        Subwf   IR_Dev,W
        Btfss   STATUS,Z
        Goto    Again
; ** If IR_But=116 (channel up) then illuminate the green LED
        Movlw   16
        Subwf   IR_But,W
        Btfss   STATUS,Z
        Goto    CH_UP
        Bsf     PORTA,1
; **If IR_BUT=117 (channel down) then illuminate the red LED
CH_UP   Movlw   17
        Subwf   IR_But,W
        Btfss   STATUS,Z
        Goto    Exit
        Bsf     PORTA,0
Exit    Call    Delay   ; Delay for 10ms (optional)
        Goto    Again
```

Listing 8. Assembler, Main code loop.

The PICBASIC PLUS version is shown in listing 9. It has exactly the same function as previously described.

```
' ** THE MAIN PROGRAM LOOP STARTS HERE **
Again:  LOW Green_LED
        LOW Red_LED                     ' Extinguish both LED's
        GOSUB IRIN                      ' Receive an IR signal
        IF IR_Dev = 255 then Again      ' Check for valid header
        IF IR_Dev <> 0 then Again       ' If not a TV DEVICE code then look again
        SEROUT PORTA.3,N9600,[IR_Dev,IR_But] ' Transmit the 2 bytes
        IF IR_But = 116 then HIGH Green_LED  ' If channel up, then green LED
        IF IR_But = 117 then HIGH Red_LED    ' If channel down, then red LED
        DELAYMS 10                      ' Delay for 10ms (optional)
        GOTO Again                      ' Do it forever
```

Listing 9. PICBASIC PLUS, Main body code.

Using the subroutine, IRIN.
Both versions of the **IRIN** subroutine may easily be incorporated into your own programs. A brief outline of the returned variables are: -

CALL or **GOSUB** IRIN
IR_DEV returns holding the DEVICE code (0..31)
IR_BUT returns holding the BUTTON code (0..127)
Both **IR_DEV** and **IR_BUT** return holding 255 if a valid header was not received.

Conclusion.
I hope that I've succeeded in illustrating that both, infrared decoding and PICmicro programming need not be the exclusive property of the *whiz kids* or *rocket scientists* among us. Assembly language will never be fully replaced by high level languages, especially if compact or critically timed code is required. But with the advent of ever increasing speeds and memory storage on the new PIC ranges being developed, this is fast becoming a non-issue. The one major advantage that assembler has, is that it's free. All the tools required for software development are downloadable from microchip's web site, there are also a plethora of datasheets and application notes, which are downloadable from the same site. But this doesn't detract from the fact that assembly language is somewhat difficult to learn and sometimes tedious to write.

Using a high level language such as PICBASIC PLUS, not only makes programming a more enjoyable experience, but opens up a whole new aspect of electronics that was previously beyond the scope of all but the most advanced hobbyist, such as I^2C, SPI serial eeprom, Analogue to Digital, Digital to Analogue interfacing among many others, the list is as long as your imagination and creativity allows. However, it's not just the hobbyist who can benefit from this remarkable language. Because, both assembler and BASIC may be freely mixed within the same program, extremely powerful and flexible programs may be written that can greatly decrease prototyping time, thus reducing the overall costs of a commercial product. After all, time is a precious commodity that should not be wasted

Above all else, have fun!
Les Johnson.