PROTON IR

Disclaimer

In order to comply with EMC directive 89/336/EEC, this product should not be used outside of a classroom or laboratory environment.

Any software supplied with this product is provided in an "as is" condition. No warranties, whether express, implied or statutory, including, but not limited to, implied warranties of merchantability and fitness for a particular purpose apply to this software. The company shall not, in any circumstances, be liable for special, incidental or consequential damages, for any reason whatsoever.

This product should not be used as a critical component in life support systems. Crownhill does not recommend the use of any of its products in life support applications where the failure or malfunction of the product can reasonably be expected to cause failure in the life support system or to significantly affect its safety or effectiveness.

In producing this, document and the associated hardware we have spent a great deal of time to ensure the accuracy of the information presented. We constantly strive to improve the quality of our products and documentation. Crownhill operates a quality system conforming to the requirements of BS EN ISO9001: 2000. Crownhill reserves the right to change the functionality and specifications of its products from time to time without prior notice.

If you should find any errors or omission in these documents or hardware, please contact us, we appreciate your assistance in improving our products and services.

Crownhill reserves the right to make changes to the products contained in this publication in order to improve design, performance or reliability. Crownhill assumes no responsibility for the use of any circuits described herein, conveys no license under any patent or other right, and makes no representation that the circuits are free of patent infringement. Charts and schedules contained herein reflect representative operating parameters, and may vary depending upon a user's specific application. While the information in this publication has been carefully checked, Crownhill shall not be liable for any damages arising as a result of any error or omission.

PICmicro[tm] is a trade name of Microchip Technologies Inc.
PROTON[tm] is a trade name of Crownhill Associates Ltd.
EPIC[tm] is a trade name of microEngineering Labs Inc.

This document was written by Les Johnson and published by Crownhill associates limited, Cambridge England, 2003

Table of Contents.

Installing the PROTON IR software.

The PROTON IR development system is shipped with a CDROM containing all relevant source code.

In order to comply with the text in this document, the source code and the PROTON IR bootloader needs to be copied into the PROTON+ Compiler's directory.

Using Windows Explorer (or whatever means you find easier), copy the IR_SAMPLES folder located on the CDROM into the Compiler's folder. Make sure that the files have not been transferred as READ-ONLY, because Windows has a peculiar habit of doing just that.

If the files have been tagged as read only, then set their attributes to not read only by right clicking the appropriate .BAS file, selecting PROPERTIES and unticking the read only checkbox.

Copy the **PROTIR_BOOT.EXE** file into the compiler's **INC** folder. The editor will then add it to the list of useable bootloaders.

If you accidentally overwrite the bootloader's firmware located within the PICmicro^{tm}, then simply re-program the PICmicro^{tm} with the **PROTIR_FIRMWARE.HEX** file.

Getting Started.

Before we go any further into the document, we'll take some time to become acquainted with the PROTON IR board, and its method of programming. We won't actually look at the electronics in any detail just yet, as there is a whole section dedicated to that subject later.

The PROTON IR is based around one of the latest PICmicro^{tm} microcontrollers. This can be considered as the work-horse of the board, but this is a dumb horse until it's told what to do via a program. In most circumstances actually getting the program into a PICmicro^{tm}, involves using a dedicated device programmer such as the EPIC^{tm}. But thanks to the magic contained in the latest PICmicro^{tm} devices, we can now program them using a simple serial interface directly from the COM port of the PC. This is commonly known as BOOTLOADING.

So what is a bootloader?

A bootloader is a program that resides in the code space of the PROTON IR's PICmicro^{tm}. It can be activated to allow additional program code to be written to and read from that same target PICmicro^{tm}. A bootloader consists of 2 elements, connected by a standard serial cable.

The first part of the bootloader is a program resident on the PROTON IR's PICmicro^{tm}. This program occupies the upper 256 words of the FLASH code space. This small program must be placed into the PICmicro^{tm} using a conventional programmer, but don't panic, it's already programmed in the PROTON IR.

The program resident in the PICmicro^{tm} communicates with the second element of the loader over a serial connection. This second program is the bootloader software that resides on the computer and is the user interface. It allows the compiled BASIC code to be programmed.

Only the code space and data space may be read and programmed on the target PICmicro^{tm}. The ID space and CONFIGURATION fuses are not accessible to the bootloader. The configuration fuses must be set at the time the actual loader program is programmed into the PICmicro^{tm}. Once they are set, they cannot be changed by the bootloader.

The bootloader software resident in the PICmicro^{tm}, intercepts the reset vector. When the PICmicro^{tm} powers up, it enters the loader's boot supervisor, this watches the serial input pin for a start bit for 200 milliseconds (ms). If it sees activity during this period, it enters the communications section of the

software to download a program. If it does not see any activity during the 200ms, it starts the user program in the PICmicro™.

The interception of the reset vector is accomplished by automatically relocating the first 4 user program words from address's 0 to 3 to a reserved place in the bootloader's code space (within the top 256 words). A jump to the bootloader is then placed at locations 0 to 3. When the loader software running on the PC reads or writes these addresses the values seen are as if the bootloader was not resident and the reset code had not been moved.

The serial pins used by the bootloader are only required when the loader is actually programming the PICmicro™. They are unattached while the downloaded user program is running on the target and may be assigned to any other task or serial baud rate.

The serial communication speed is set at 19200 baud. The bootloader program resident in the PICmicro™ can easily communicate at this speed with an oscillator frequency from 4MHz. This oscillator frequency is determined at the time the loader code is programmed into the target PICmicro™. The target PICmicro™ must then only be run at this frequency in order to be able to communicate with its matching part running on the PC. The bootloader uses no PICmicro™ resources while the user program is running. All the data memory, RAM, and I/O pins are available to the user program.

However, there are a few considerations that should be noted when writing programs that will be loaded by the bootloader. The first is that the bootloader takes over at power up and any subsequent resets. Any time the program vectors through the reset address, the loader becomes active and watches the RX pin (PORTB.7) for any activity. If there is any action on this pin, the loader will start, and the user program will not execute. Even if there is no activity on this pin, the start of the user program will be delayed by the 200 milliseconds while the bootloader is watching the RX pin.

Another consideration is the fact that the configuration fuses are not alterable by the bootloader. If some programs require the use of the Watchdog Timer and others don't, then a reprogram of the PROTON IR's PICmicro™ will be necessary. The same is true for the Power up Timer, Brownout Detect Enable etc. The standard programmed defaults are Watchdog Timer OFF, Powerup Timer ON, Brownout Detect Enable OFF and XT oscillator.

The configuration fuses for code protection CANNOT be enabled. The bootloader needs to be able to freely read and write to the PICmicro's code and data space. Therefore, the device cannot be code protected.

The bootloader is primarily aimed at development work, any final products that require code protection must be programmed in the conventional way.

The bootloader software occupies the last 256 words of code space. A compiled program is written starting at location 0 and grows upward so the loader's position in memory is not noticeable. You must make sure that the program code does not attempt to enter the upper 256 words of code space, or the PROTON+ compiler will report an error. The bootloader inserts its own code at the reset vector and automatically relocates the user's reset code to an area reserved within the top 256 words of memory. Normally, these locations contain a jump to the start-up routine for the user program. However, since the user code is no longer situated at these locations, the user program should not attempt to jump to, or call any routine within the code area between 0 and 3.

Programming the PROTON IR.

Programming the PROTON IR is simplicity itself, thanks to the resident bootloader (see above). And because this document, and its code listings are based around the PROTON+ compiler, we'll take a step by step approach centred around the compiler's IDE.

Step 1.
Run the PROTON+ compiler, and load the program **PR_FLASH.BAS**, this is located in the PROTON+ compiler's directory, inside the **IR_SAMPLES** folder. Or alternatively, you can type it in from the listing below: -
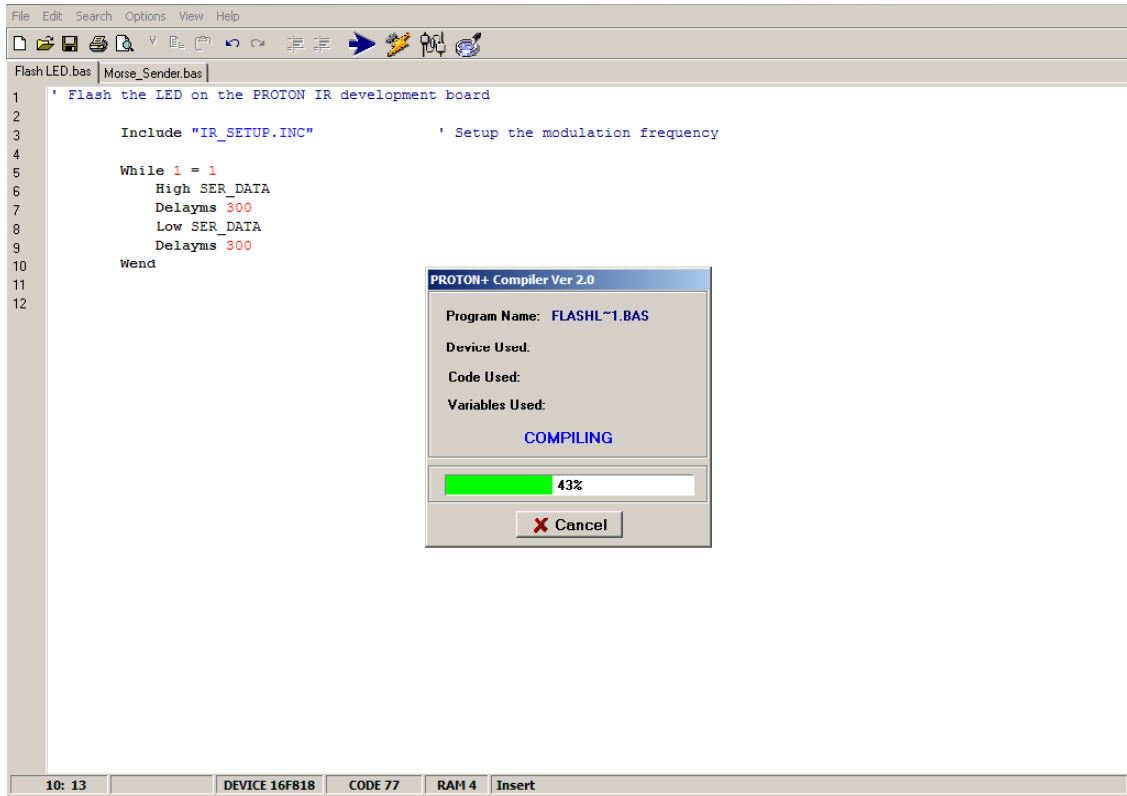
```
' Flash the LED on the PROTON IR development board

Include "IR_SETUP.INC"          ' Setup the modulation frequency

While 1 = 1
High SER_DATA
Delayms 300
Low SER_DATA
Delayms 300
Wend
```

Compile the program, by clicking on the **COMPILE** icon located on the toolbar, and you should see the screen shown overleaf.
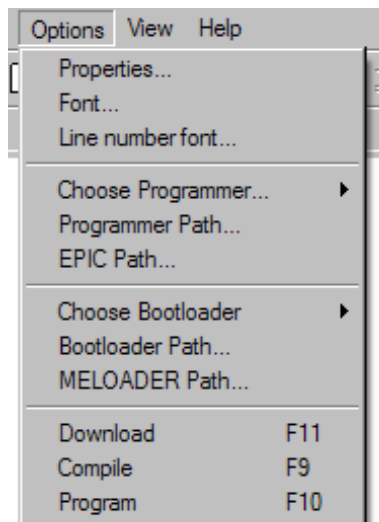
If no errors were produced while compiling, the program is ready for downloading to the PROTON IR board, but we need to choose the method from the IDE.
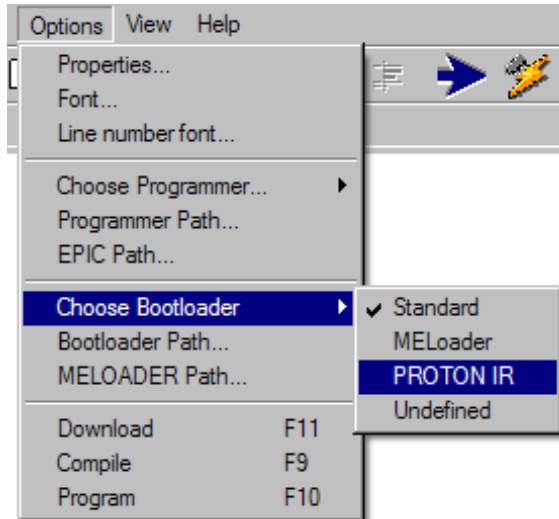
Step 2.
Under the **OPTIONS** menu, you'll see a menu option for **CHOOSE BOOT-LOADER**.

The default bootloader is the standard type shipped with the compiler, but the PROTON IR uses a different type, therefore place the mouse pointer over the **CHOOSE BOOTLOADER** item, and another small menu will appear, this shows what bootloaders are available for use (see overleaf).

Click on the **PROTON IR** option, and the menu will close. The compiler's IDE is now set up so that whenever the download icon is [icon] selected, the PROTON IR's bootloader will be used.

Step 3.

Connect the PROTON IR programming board to the PC using the serial cable supplied, and connect the power. Then place the PROTON IR board into its programming cradle.
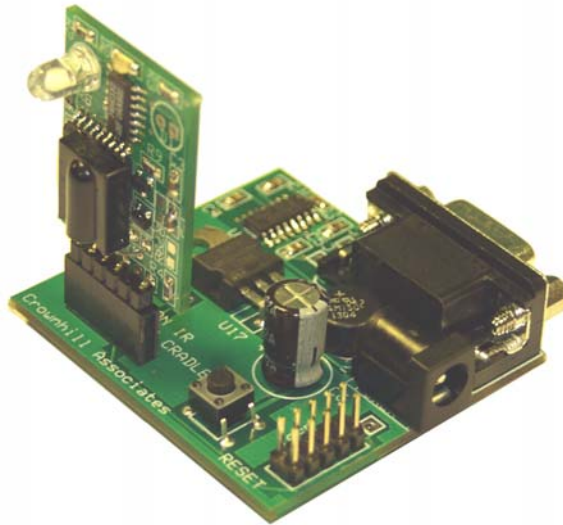
Now click on the **DOWNLOAD** icon (shown above). You will be greeted with the window shown below: -
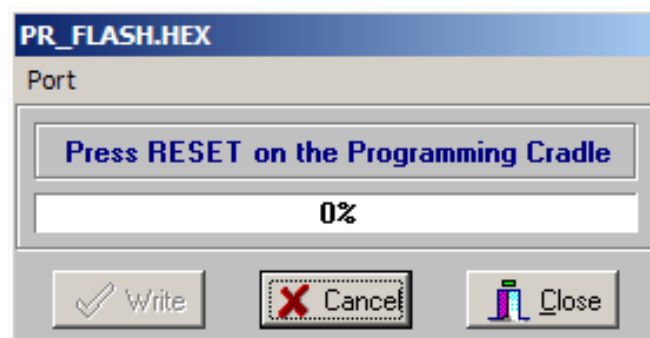
On some occasions, you will not be required to press the RESET button, and in this case, the program will be downloaded immediately.

If the program was successfully downloaded to the PROTON IR board, then the red LED at the top of the PROTON IR should now be flashing at a rate of one flash per 300ms. You've now successfully programmed the PROTON IR board, easy wasn't it? All future discussions and programs will be downloaded the same as just explained.

However, if the program failed to download and shows the below window: -
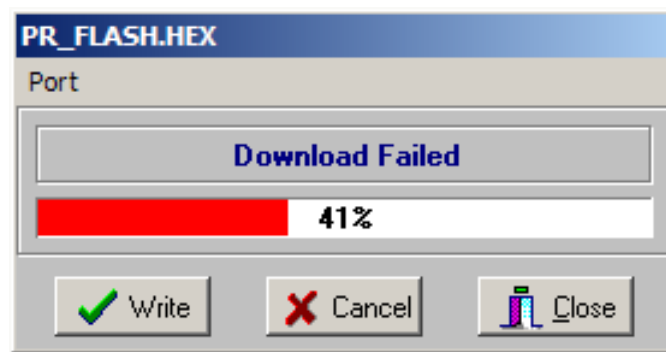
then don't panic, simply click on the WRITE button, and start the process again. The PROTON IR cannot easily be damaged if it's sitting in it's cradle.

Now that we know how to load a program into the PROTON IR board, we can have some fun with it. Read on!

What is Infrared Light ?

What we humans perceive as visible light, is a tiny portion of the electromagnetic spectrum. Waves in the electromagnetic spectrum vary in size from very long radio waves the size of buildings, to very short gamma-rays smaller than the size of the nucleus of an atom.
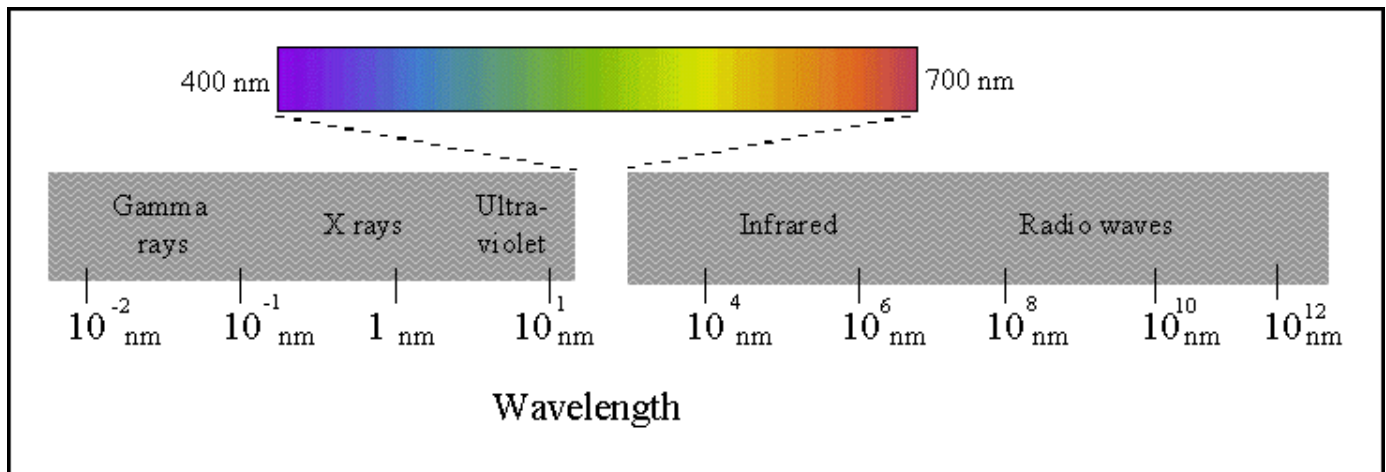

The section of the electromagnetic spectrum that we're interested in is located just before visible red light occurs, at a wavelength of approx 800 to 900nm (Billionth of a Metre). The term infra means 'below', so infra red light means below red light. Infra red light is the preferred medium of remote controls because it is invisible to humans, and therefore does not pollute other light sources. However, the Sun, and man-made light sources encroach in the near infra red light spectrum to a certain degree, so we must make allowances for this, and try and eliminate a potential nuisance. The way we do this is through modulation.

Modulation.

Modulation is a method of encoding digital (or analogue) signals on a different waveform (the carrier signal). Once encoded, the original signal may be recovered by an inverse process, demodulation. Modulation is performed to adapt the signal to a different frequency range (and medium) than that of the original signal. The carrier signal in our case is infrared light, but the same analogy is commonly used for radio waves, such as A.M (Amplitude modulation), and F.M (Frequency Modulation). The method used for infrared remotes is mostly a form of Pulse Coded Modulation (PCM). This method produces a digital signal of 1's and 0's, depending on whether the modulated carrier is present, or not. This is easier to explain diagrammatically.

Shown below are two diagrams that illustrate the modulation method used by infrared remote controls handsets.

SWITCH

Oscillator

LED

LED Output
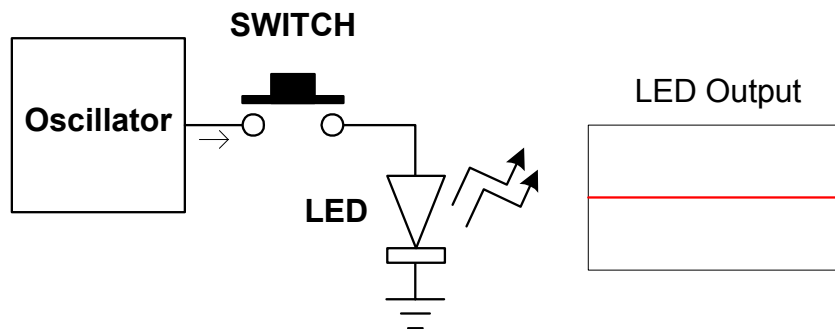
The oscillator block shown above is assumed to be running continuously, however, is blocked from illuminating the LED by the switch, which is in the **OFF** position.

SWITCH

Oscillator

LED

LED Output
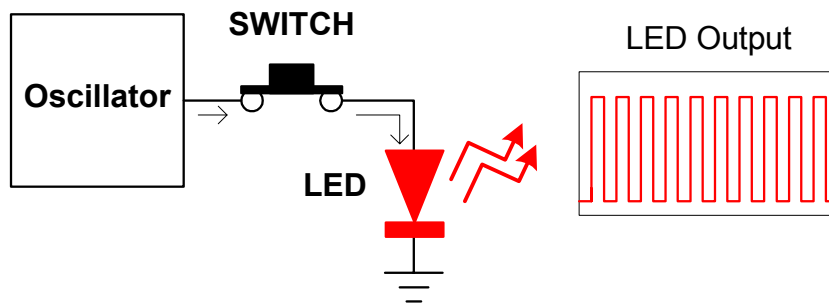
When the switch is closed (**ON** position), the oscillator's output frequency, modulates the LED. Therefore the LED is not illuminated continuously, but is being chopped at the frequency dictated by the oscillator. In reality, this modulating frequency ranges from around 36KHz to 40KHz depending on the make and model of the remote transmitter/receiver combination. On the receiving side of infrared communications is a special sensor that produces an output when modulated infrared light of a certain frequency and wavelength (see electromagnetic spectrum) is detected, but ignores non modulated infrared light (more on this later).

In the above examples, we have two states; ON or OFF. This is good for demonstration, but not much use in a practical situation. What's required is a form of protocol in order to turn a sequence of ONs (1's) and OFFs (0's) into useable information, such as letters, numbers etc.

A well established protocol that springs to mind is the Morse Code. Invented by Samuel Morse (*pictured right*) back in the early 19th century. This consists of a sequence of ON-OFF signals represented by the well known DAH-DIT sounds.

PROTON INFRARED

We can demonstrate modulating the PROTON's infrared LED using the Morse code very simply, and in doing so, gain further insight into how we can elaborate on it.

The Morse code involves two states, arranged in a specific sequence that represent letters and numbers. The table below shows the sequences of dots and dashes that make up the common alphabet.

Morse Code Alphabet		
A ●■	N ■●	0 ■■■■■
B ■●●●	O ■■■	1 ●■■■■
C ■●■●	P ●■■●	2 ●●■■■
D ■●●	Q ■■●■	3 ●●●■■
E ●	R ●■●	4 ●●●●■
F ●●■●	S ●●●	5 ●●●●●
G ■■●	T ■	6 ■●●●●
H ●●●●	U ●●■	7 ■■●●●
I ●●	V ●●●■	8 ■■■●●
J ●■■■	W ●■■	9 ■■■■●
K ■●■	X ■●●■	FullStop ●■●■●■
L ●■●●	Y ●■●■	Comma ■■●●■■
M ■■	Z ■■●●	Query ●●■■●●

The key ingredient, that needs to be added to turn ON and OFF signals into a distinguishable piece of data is TIME. Each ON/OFF state has a predetermined length in which to be in. For example, the Morse code DASH is three times as long as a DOT. Shown below are the timings based on 12 Words per Minute (WPM).

DOT (DIT).......33ms (milliseconds).
DASH (DAH)...99ms (Three times the length of a DOT).
TIME between DOTs and DASHES is the length of a DOT. i.e. 33ms.
TIME between two CHARACTERS is the length of three DOTS. i.e. 99ms.
SPACES between WORDS is 231ms (Seven times the length of a DOT).

As you can see from the list above, the elementary measure of time is given by the DOT. Every other part of the protocol is a measure of this. A DOT can be any length from 1 millisecond to 1 second, but the data is still distinguishable because a DASH will always be three times longer. We can think of the DOT as being an OFF state, and the DASH as being an ON state, but we now have a third state, that of NO SIGNAL for a predetermined time. i.e. SPACES, and the time between individual DOTs and DASHES.

We now have all the ingredients for a protocol, but we must apply some rules to be able to transmit data that can be decoded. If no rule was applied, we would be able to send individual dots and dashes, but not understand what was being sent. That's where the Morse alphabet comes into it's own (see previous page).

In the Morse alphabet, a sequence of DOTs and DASH's is used to build a letter, or number. For example: -
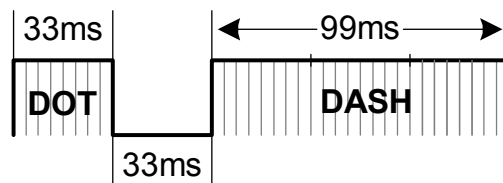
DOT – DASH represents the letter A.

In digital terms this will be : -

Modulated carrier ON for 33ms... (DOT).
NO Modulated carrier for 33ms.
Modulated carrier ON for 99ms... (DASH).

The above list can be displayed in a more informative timing diagram, shown below.

Letter A

As you can see from the above diagram, the sequence of dots and dash's is starting to look more like a conventional timing diagram. We'll be seeing more of the above timing diagrams later in the document, to explain other serial protocols.

To group together a string of letters to form a word is a simple matter of applying the timing rules. For example to send the word "OK", we'd look at the alphabet and see that letter "O" is a sequence of three DASH's, then delay 99ms (3 DOT lengths), then the letter "K", which is a DASH, DOT, DASH sequence.

Again, in digital terms, this would be: -

Modulated carrier ON for 99ms... (DASH). \
NO Modulated carrier for 33ms.   \
Modulated carrier ON for 99ms... (DASH).   **Letter "O"**
NO Modulated carrier for 33ms.   /
Modulated carrier ON for 99ms... (DASH). /
NO Modulated carrier for 99ms... (Inter-character space).

Modulated carrier ON for 99ms... (DASH). \
NO Modulated carrier for 33ms.   \
Modulated carrier ON for 33ms... (DOT).   **Letter "K"**
NO Modulated carrier for 33ms.   /
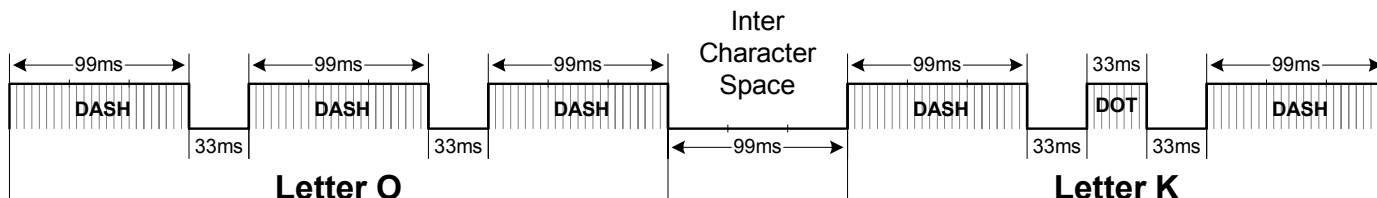Modulated carrier ON for 99ms... (DASH). /

And the timing diagram is shown below.

Now we're really starting to see a pattern that can be decoded either by a human operator or by a machine (i.e. computer).

To digress from our subject matter for a few minutes, it turns out that the Morse alphabet is not a random collection of dots and dash's as we all thought (well, as I thought anyway!), but a carefully calculated strategy, which is just what is required for a successful protocol.

It is possible to form the Morse code for all letters by using the table above. You start from the top centre. Whenever you go to left in the table, you add a DOT "." to the code and whenever you go to right you add a DASH "-" to the code. For example, the letter "K" is formed by going to RIGHT-LEFT-RIGHT from which we get its code DASH - DOT - DASH. I'm not totally convinced that this was the intention of Samuel Morse, or a phenomenon that just happens to fit the pattern, but it does work.

Sending a Signal.
After that small digression, we can now get back to the subject of modulation, and protocols. Run the PROTON+ compiler, and load the program **MORSE_SEND.BAS**, this is located in the PROTON+ compiler's directory, inside the **IR_SAMPLES** folder. The full listing of the program is shown on the next page.

```
' Program MORSE_SEND.BAS
' Send Morse code signals via infrared transmission

Include "IR_SETUP.INC"                    ' Setup the modulation frequency

Dim MORSE_LOOP as Byte
Dim MESSAGE_OFFSET as Word
Dim CHARACTER as Byte
Dim DOT_DASH as Byte


Symbol DOT = 33                    ' DOT time of 33ms
Symbol DASH = DOT * 3                      ' DASH timing (3 * DOT)
Symbol INTER_WORD = DOT * 7        ' Inter WORD timing (7 * DOT)
Symbol INTER_CHARACTER = DOT * 3 ' Inter CHARACTER timing (3 * DOT)
Symbol NUL = 0                             ' NUL terminator


Goto OVER_MORSE_SUBROUTINES       ' Jump over the subroutines


'----------------------------------------------------------------
' Message to send goes here, terminated by a NUL
MESS: LDATA "HELLO WORLD HOW ARE YOU"
      LDATA NUL
'----------------------------------------------------------------
' Sends the character held in variable "CHARACTER"
SEND_MORSE:
While 1 = 1
DOT_DASH = LREAD (CHARACTER * 6) + MESSAGE_OFFSET
Inc MESSAGE_OFFSET
If DOT_DASH = NUL Then Return
High SER_DATA
Delayms DOT_DASH * 2
Low SER_DATA
Delayms DOT * 2
Wend


'------[MAIN PROGRAM LOOP]-------------------------------------------
OVER_MORSE_SUBROUTINES:

START_OVER_AGAIN:
Delayms INTER_WORD * 2                    ' Send a SPACE between words
MORSE_LOOP = 0
While 1 = 1                               ' Create an infinite loop
' Read each character to send from LDATA
CHARACTER = LREAD MESS + MORSE_LOOP
If CHARACTER = NUL Then START_OVER_AGAIN ' Exit if message ended
If CHARACTER = " " Then                   ' Is the character a space ?
Delayms INTER_WORD * 2                     ' Send a SPACE between words
Goto INCREMENT_CHARACTER                   ' Jump over the rest of the
code
Endif
If CHARACTER >= "A" AND CHARACTER <= "Z" Then ' Is the character a letter
MESSAGE_OFFSET = LETTER_A        ' Point to the letter ldata tables
CHARACTER = CHARACTER - "A"      ' Remove the ASCII coding
Gosub SEND_MORSE                 ' Send the character
```

```
Endif
If CHARACTER >= "0" AND CHARACTER <= "9" Then ' Is the character a number
MESSAGE_OFFSET = NUMBER_0          ' Point to the number ldata tables
CHARACTER = CHARACTER - "0"        ' Remove the ASCII coding
Gosub SEND_MORSE                   ' Send the character
Endif
INCREMENT_CHARACTER:
Delayms INTER_CHARACTER * 2 ' Implement a space between two characters
Inc MORSE_LOOP                     ' Move up the list of characters
Wend


'-----------------------------------------------------------------
' Morse dots and dash's data
NUMBER_0:  LDATA DASH,DASH,DASH,DASH,DASH,NUL
NUMBER_1:  LDATA DOT,DASH,DASH,DASH,DASH,NUL
NUMBER_2:  LDATA DOT,DOT,DASH,DASH,DASH,NUL
NUMBER_3:  LDATA DOT,DOT,DOT,DASH,DASH,NUL
NUMBER_4:  LDATA DOT,DOT,DOT,DOT,DASH,NUL
NUMBER_5:  LDATA DOT,DOT,DOT,DOT,DOT,NUL
NUMBER_6:  LDATA DASH,DOT,DOT,DOT,DOT,NUL
NUMBER_7:  LDATA DASH,DASH,DOT,DOT,DOT,NUL
NUMBER_8:  LDATA DASH,DASH,DASH,DOT,DOT,NUL
NUMBER_9:  LDATA DASH,DASH,DASH,DASH,DOT,NUL
LETTER_A:  LDATA DOT,DASH,NUL,NUL,NUL,NUL
LETTER_B:  LDATA DASH,DOT,DOT,NUL,NUL,NUL
LETTER_C:  LDATA DASH,DOT,DASH,DOT,NUL,NUL
LETTER_D:  LDATA DASH,DOT,DOT,NUL,NUL,NUL
LETTER_E:  LDATA DOT,NUL,NUL,NUL,NUL,NUL
LETTER_F:  LDATA DOT,DOT,DASH,DOT,NUL,NUL
LETTER_G:  LDATA DASH,DASH,DOT,NUL,NUL,NUL
LETTER_H:  LDATA DOT,DOT,DOT,DOT,NUL,NUL
LETTER_I:  LDATA DOT,DOT,NUL,NUL,NUL,NUL
LETTER_J:  LDATA DOT,DASH,DASH,DASH,NUL,NUL
LETTER_K:  LDATA DASH,DOT,DASH,NUL,NUL,NUL
LETTER_L:  LDATA DOT,DASH,DOT,DOT,NUL,NUL
LETTER_M:  LDATA DASH,DASH,NUL,NUL,NUL,NUL
LETTER_N:  LDATA DASH,DOT,NUL,NUL,NUL,NUL
LETTER_O:  LDATA DASH,DASH,DASH,NUL,NUL,NUL
LETTER_P:  LDATA DOT,DASH,DASH,DOT,NUL,NUL
LETTER_Q:  LDATA DASH,DASH,DOT,DASH,NUL,NUL
LETTER_R:  LDATA DOT,DASH,DOT,NUL,NUL,NUL
LETTER_S:  LDATA DOT,DOT,DOT,NUL,NUL,NUL
LETTER_T:  LDATA DASH,NUL,NUL,NUL,NUL,NUL
LETTER_U:  LDATA DOT,DOT,DASH,NUL,NUL,NUL
LETTER_V:  LDATA DOT,DOT,DOT,DASH,NUL,NUL
LETTER_W:  LDATA DOT,DASH,DASH,NUL,NUL,NUL
LETTER_X:  LDATA DASH,DOT,DOT,DASH,NUL,NUL
LETTER_Y:  LDATA DOT,DASH,DOT,DASH,NUL,NUL
LETTER_Z:  LDATA DASH,DASH,DOT,DOT,NUL,NUL
```

After a successful compilation, download the program to the PROTON IR board (see the first section of this document), and the red LED on the PROTON IR board will start flashing out the Morse code for the text "HELLO WORLD HOW ARE YOU". As well as the red LED being illuminated, the infrared diode/s are also flashing out the message, but of course we can't see them directly, because the light is invisible to human eyes (see Electromagnetic Spectrum).

I promise we'll look more closely at how the PROTON IR modulates its infrared LED/s at 38KHz soon, but for now simply bask in the knowledge that everything's going to plan. But how do we know the infrared diodes are flashing if we can't see them?. We need a receiver that can see modulated infrared light and do some conversion for us.

Receiving a Signal.

If you own a PROTON development board, then you can use that for infrared reception, as it comes equipped with a sensor (we'll get to using this in a minute). But for now, we'll concentrate on using another PROTON IR board for reception. Load the program **MORSE_REC.BAS** into the compiler, this is also located in the PROTON+ compiler's directory, inside the **IR_SAMPLES** folder. The full listing of the program is shown below: -

```
' Receive a signal from another PROTON IR board
' And flash the LED in sympathy with the incoming signal

Device = 16F819        ' PICmicro device on the PROTON IR
XTAL = 4               ' Crystal frequency of 4MHz


Symbol IR_SENSOR = PORTB.0 ' Pin where the IR sensor is connected
Symbol LED = PORTB.1
Delayms 500            ' Wait for the PICmicro to stabilise
ALL_DIGITAL = TRUE     ' PORTA to all digital

TRISB.2 = 0            ' CCP1 (PortB.2 = Output)
PR2 = 25               ' Set PWM Period for approximately 38KHz
CCPR1L = 13            ' Set PWM Duty-Cycle to 50%
CCP1CON = %00001100    ' Mode select = PWM
T2CON = %00000100      ' Timer2 ON + 1:1 prescale


'------[MAIN PROGRAM LOOP]--------------------------------------
While 1 = 1            ' Create an infinite loop
LED = ~IR_SENSOR       ' Invert the incoming IR signal
Wend
```
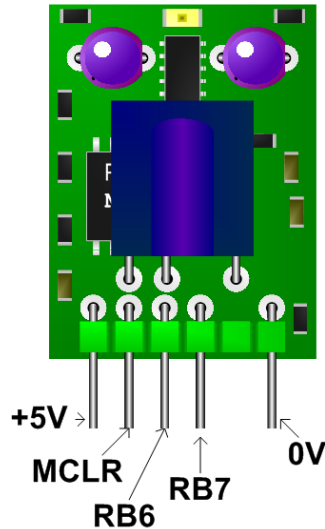
Again, after a successful compilation, download this program to the second PROTON IR board, we now need to apply some power. Shown overleaf are the pin outs for the PROTON IR board.

It is important that the correct assignments are observed, as the wrong polarity, or voltages over 5 Volts can, and will, cause damage to the board.

MCLR pin is used to RESET the PROTON IR board.
RB6 pin is used for serial data OUT.
RB7 pin is used for serial data IN.

Once power is applied to both the transmitting and receiving PROTON IR boards, both LEDs will be flashing in unison. As can be seen from the receiver BASIC code, the illumination of the LED is directly related to the incoming signal, therefore, is indicating a signal entering the IR sensor.

Receiving on a PROTON development board.

As promised earlier, here is the code that receives the incoming infrared data on the PROTON development board.

Connect the link on jumper **J3**, to bring the IR sensor into play, and load the program **PROTON_MREC.BAS**, this can be found alongside all the other programs in the **IR_SAMPLES** directory. The program is also shown below: -

```
' Receive a signal from a PROTON IR board
' And flash the LED in sympathy with the incoming signal
' This program is intended for the PROTON Development Board.

Include "PROTON_4.INC"
Symbol IR_SENSOR = PORTC.0 ' Pin where the IR sensor is connected
Symbol LED = PORTD.7        ' LED to flash in sympathy
'------[MAIN PROGRAM LOOP]-------------------------------------------
Delayms 500                      ' Wait for the PICmicro to stabilise
OUTPUT LED                   ' Set the led pin to an output
While 1 = 1                      ' Create an infinite loop
LED = ~IR_SENSOR          ' Invert the incoming IR signal
Wend
```

How does the PROTON IR work?

We've now established a communication pathway for information from one PROTON IR board to another using modulated infrared light, and we've established that a protocol is essential for it all to work. So we'll leave protocols alone (for the time being), and concentrate on how the light is actually modulated by the PROTON IR, and how it's received.

We'll look at the receiving side first, as this is the key ingredient that dictates what frequency; and type of modulation is required.

Receiving Circuit.

To receive a particular wavelength of light, and demodulate it in order to give a signal when only modulated light of a certain frequency is detected, could be designed using discrete components, but luckily, ready made infrared (IR) sensors are plentiful and very inexpensive, thanks, in part, to being used in TV sets, Videos, HI-FIs, and most Camcorders now. These sensors come in all shapes and sizes, but the more common types have the appearance of the one shown below: -
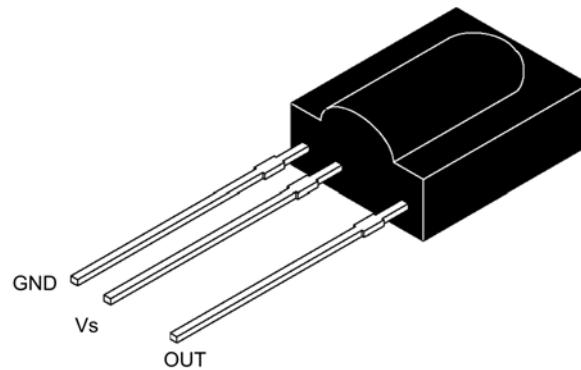
Looking from the outside, these devices are similar in appearance to a standard photodiode, but internally, they are crammed with electronics, as the block diagram (shown below) of one of these devices illustrates: -
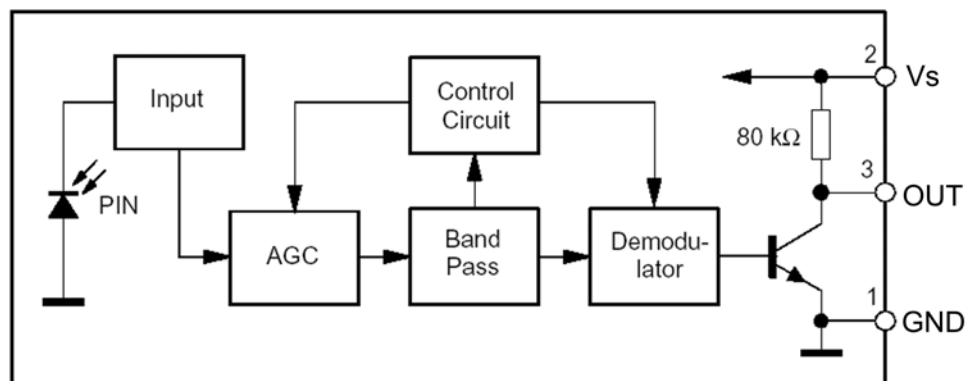
Each of the internal blocks shown, contains several transistors and resistors. Which amount to quite a complex circuit (thank goodness for miniaturisation).

The infrared sensors may be similar in appearance, but differ in a very important element. In that each type of sensor comes in a range of modulation frequencies that they are most sensitive to. For example, on the PROTON IR board, we're using a sensor that is more sensitive to the modulation of 38KHz, but some range from 36KHz to 40KHz. This is dependant on the application that they are intended to be used in, and is why we needed to look at the receiver before we tailored the infrared diode's modulation frequency.

The output from the infrared sensor is TTL compatible, which means we can attach it directly to the PICmicro's pin, as shown below: -
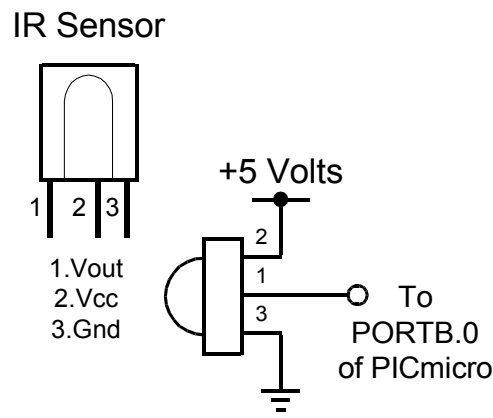
A very significant aspect of the infrared sensor (and most IR sensors of this type), is that the output is active low. Which means that it's idle state (no modulated light detected) is logic high, while a valid signal causes the output to be logic low. i.e. 0 Volts. This is crucial to know from a software point of view, and if you examine the Morse receiving programs, you'll see that the IR_SENSOR input is complemented in order to invert it before it is output to the LED: -

```
LED = ~IR_SENSOR
```

This is essential to remember, otherwise we wouldn't know if we were receiving a 1 or a 0.

Transmitting Circuit.

Now we understand the requirements of the infrared sensor used, we can design a hardware/software combination that will modulate the infrared diode/s to the correct frequency, and send the appropriate logic level for a 1 or a 0.

The PICmicro™ used on the PROTON IR board, has a hardware feature that is invaluable in performing modulation. The hardware feature I'm referring to is the Hardware Pulse Width Modulation (HPWM) MSSP module.

Once setup, the HPWM pin will continue to output a frequency, without slowing down, or interfering with the main BASIC code.

PORTB.2 on the 16F819 also functions as the HPWM output pin (CCP1) when not being used for normal I/O operations. Setting up the HPWM frequency is a simple matter of loading some of the PICmicro's hardware registers.

Registers TMR2 (Timer2), and PR2 (TMR2's PERIOD Register) are used to establish the period of the PWM output. The timing diagram below helps illustrate how this works.
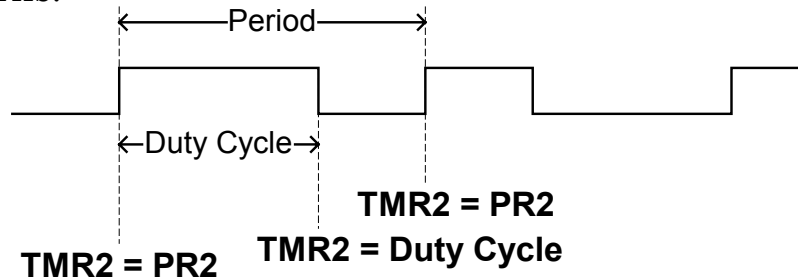
A PWM output has a time base (period) and a time that the output stays high (duty cycle). The frequency of the PWM is the inverse of the period, which is (1/period).

The above timing diagram shows the period from TMR2 = PR2 to TMR2 = PR2. This represents one complete cycle. To establish the period required to generate a frequency of 38KHz, use the calculation; 1/38KHz, or 1/38,000. Which results in 0.000026316 (26.3uS).

We now know that in order to generate a frequency of 38KHz, we'll need each period to be approximately 26 microseconds (us). A 50% duty cycle would require 13uS high, and 13uS low on the I/O pin.

Frequency (f) and Period (P) are inversely proportional: -

$f = 1/P$

or

$P = 1/f.$

To calculate the value to be loaded into PR2: -

PR2 = (4MHz / (4 * TMR2 prescale value * 38KHz)) - 1.

We'll use a 1:1 prescale ratio value for TMR2, which makes the calculation: -

        4 * 1 * 38,000 = 152,000
        4,000,000 / 152,000 = 26.315
        26.315 - 1 = 25.315

We can't load a fractional value into a register, so we'll strip the values after the decimal point (Truncate it), and load 25 into PR2, and accept the small amount of error produced.

The catch with the HPWM module is that as frequency increases, so resolution decreases. Note: that resolution refers to the resolution of the duty cycle, and not the actual frequency of the PWM.

To calculate the resolution for a given frequency, we use the calculation below: -

$$\text{Resolution} = \frac{\log\left(\frac{\text{FOSC}}{\text{FPWM}}\right)}{\log\left(2\right)} \text{ bits}$$

This shows how to find the maximum PWM resolution (in bits) for a given PWM frequency, with our selected oscillator frequency.

The PROTON IR board uses a 4MHz oscillator, so we need to calculate:-

        Log (4MHz/38KHz) / Log(2) to find our maximum resolution in bits: -

        Log (4,000,000/38,000) = 2.022
        Log(2) = .301

So the maximum resolution is found to be 2.022/.301 = 6.7 bits. So we'll round this down to 6 bits of resolution.

To setup the duty cycle of 50%, there are two registers that need to be loaded. CCPRL1 contains the eight (most significant bits), and CCP1CON <4:5> (CCP1CON bits 4 and 5) contain the two (least significant bits) of the duty required.

Since we will only have a maximum of 6-bits resolution, we only need to load the CCPR1 register. CCP1CON register's bits 4 & 5 will be loaded with 0's.

To calculate the value that needs to be loaded into the CCPRL1 register for 38KHz at 4MHz with a 50% duty cycle: -

Value for CCPRL1 = (PR2 + 1) * TMR2 prescale * 50% Duty Cycle

or

(25 + 1) * 1 * 0.50 = 26 * 0.50 = 13

So we know that we need to load CCPRL1 with 13 for a 50% duty cycle.

We now have all the pieces of information required to produce the correct modulating frequency for our infrared LED/s. And the BASIC code enabling us to put the pieces together is shown below: -

```
PR2 = 25                ' Set PWM Period for approximately 38KHz
CCPR1L = 13             ' Set PWM Duty-Cycle to 50%
CCP1CON = %00001100     ' Mode select = PWM
T2CON = %00000100       ' Timer2 ON + 1:1 prescale ratio
TRISB.2 = 0             ' CCP1 PORTB.2 = Output
```

Notice that register CCP1CON is loaded with a binary value of 00001110, which sets bits 2 and 3. These are named CCP1M3 and CCP1M2, and configure the MSSP's PWM mode. Register T2CON bit 2 is set to start the timer. Bits 1 and 0 are left clear for a prescaler ratio of 1:1. Clearing TRISB.2 sets up the CCP1 pin as an output which is necessary to have the pin produce the 38KHz PWM frequency.
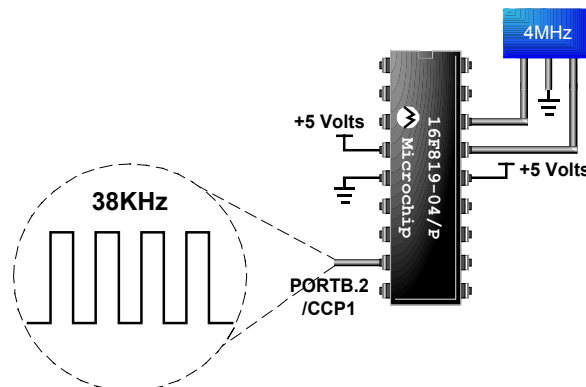
If we were to build the circuit above, and load the program shown into the PICmicro[tm], we would see a clean, uniform square wave on the CCP1 pin (PORTB.2).

Now that we have our 38KHz modulation frequency, we need a circuit that will combine the modulated carrier and data signal. i.e. the ONs, and OFFs.

In the earlier discussion on modulation, I showed how a switch can be used to 'gate' the oscillator's output to the LED. However, this is impractical in an actual application, so we'll use an electronic equivalent to a switch.

The switch we'll use is the simple logic gate known as a NAND (Not And). But the NAND gates used also contain a Schmitt circuit that will help reduce any spurious oscillations, thus maintain the clean square wave output.

A NAND gate will maintain a logic 0 (low) at its output, only if both inputs are at logic 1 (high). The four states available are shown below.
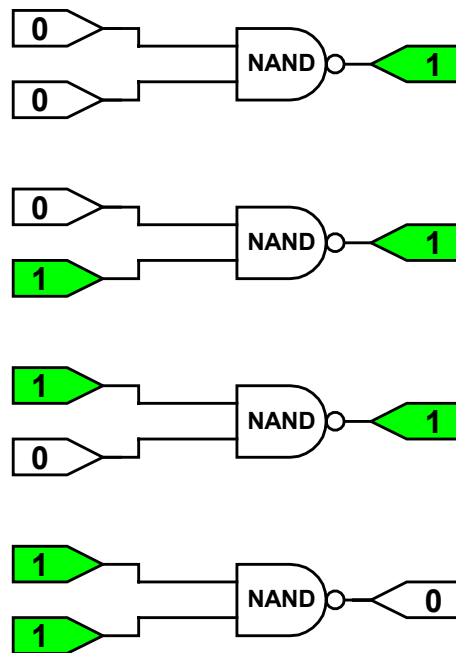
Because we're driving a maximum of two infrared LEDs, we also need a circuit that can supply the current requirements that these components demand, which can reach 100mA each. So we need a buffered output; based around a single PNP transistor. Shown below: -
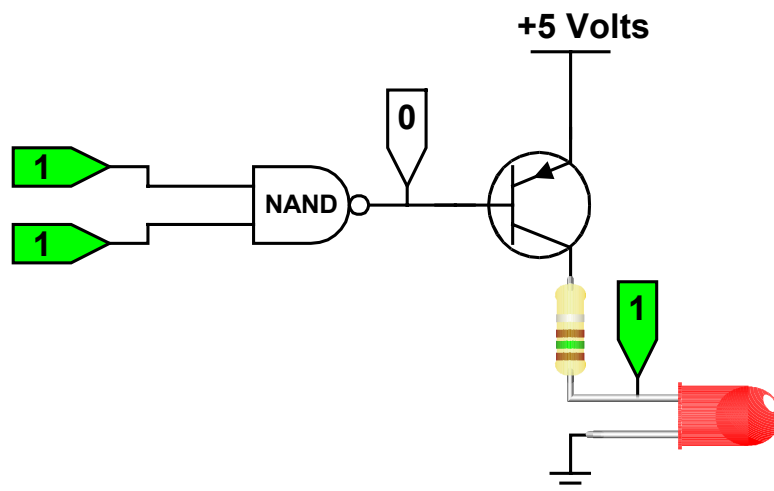
The PNP buffer has the effect of inverting the output of the NAND gate, thus producing an AND gate. Which has the opposite logic output of the NAND, in that two highs on the inputs will produce a high on the output, while a combination of highs and lows on the inputs, will always produce a logic low on the output.

If we connect one of the inputs of the NAND gate to the CCP1 pin (PORTB.2) of the PICmicro™, then this pin will be continually switched from low to high, at a rate of 38000 times per second. i.e. 38KHz. However, this will not be transferred to the output until the other input is held high. This is now our switch. By applying a controlled stream of highs and lows to the input of the NAND gate, we can transmit modulated data.

The complete circuit for the PROTON IR board is shown below: -
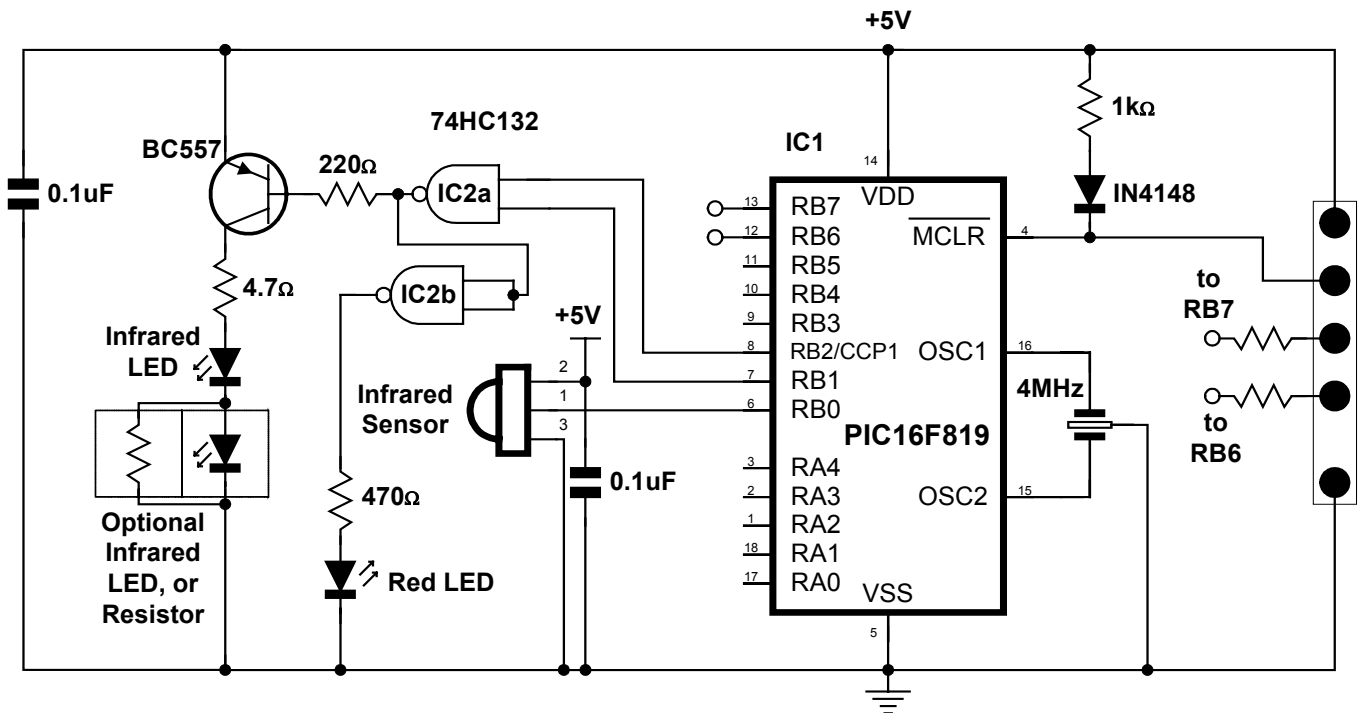
As can be seen from the circuit above, three important PICmicro™ pins are brought to the outside world. Pins RB6, and RB7, are used for the serial communications of the built-in bootloader, but are also the PICmicro's programming pins (Clock and Data). The MCLR pin (also known as VPP), controls the PICmicro™ RESET, and is also responsible for placing the device into programming mode. The combination of these three pins allows the PROTON IR board to be also programmed using a conventional device programmer. Thus offering a measure of protection against the code inside the PICmicro™ being copied by unauthorised persons. A process that is not offered by serial bootloaders. See "How does the Programming Cradle Work ?", located at the end of this document.

Starting the good stuff.

With all the information fresh in our minds, we can now start to look at more conventional uses for infrared data transfers. Morse code transmission makes a good example, but is impractical in 'real world' situations because it was not conceived for translation by a machine. There are methods that work well for Morse decoding, but I'll leave that up to the reader for further investigation. Instead, we'll look at how conventional remote control handsets operate. The types found in Televisions, Videos etc.

There are many types of remote control protocols, some complex, and some more simple in design. We'll focus on the two most popular types, the RC5 protocol designed by Phillips, and adopted by many electronics manufacturers, and the Sony SIRC protocol, which is both simple to implement, and effective in it's use.

A smidging of History.

Remote control for televisions did not simply appear overnight, someone had to invent it, and that someone was Robert Adler (shown right). His work was an important step towards the true 'couch potato' and it's only fitting that his story is told here.

Robert Adler was born in Vienna, Austria, in 1913. After receiving his doctorate in physics at age 24 from the University of Vienna (1937), he became engaged in patent work there, and later went to England. After the war broke out, he emigrated to America, and in 1941 he found work in the Research division of Zenith Electronics Corporation, Chicago.

After the war, Adler turned his attention specifically to television technology. One early invention of Adler's was the "gated-beam" thermionic valve, which eliminated a great deal of sound interference in television receivers at one stroke, thus reducing costs as well. Adler also led the team that invented a special synchronising circuit that improved reception at the fringes of a television station's broadcast area.

But Adler's greatest triumph was the wireless remote control. The first machines to be operated by remote control were used mainly for military purposes. Radio-controlled motorboats, developed by the German navy, were used to ram enemy ships in WWI. Radio controlled bombs and other remote control weapons were used in WWII. Once the wars were over, scientists experimented to find non-military uses for the remote control. In the late 1940's automatic garage door openers were invented, and in the 1950's the first TV remote controls were used.

The first TV remote control, called "Lazy Bones," was developed in 1950 by Zenith Radio Corporation. Lazy Bones used a cable that ran from the TV set to the viewer. A motor in the TV set operated the tuner through the remote control.

Although customers enjoyed having remote control of their television, they complained that people tripped over the unsightly cable that meandered across the living room floor. Zenith engineer Eugene Polley invented the "Flashmatic," which represented the industry's first wireless TV remote. Introduced in 1955, Flashmatic operated by means of four photo cells, one in each corner of the TV cabinet around the screen. While it pioneered the concept of wireless TV remote control, the Flashmatic had some limitations. It was a simple device that had no protection circuits and, if the TV sat in an area in which the sun shone directly on it, the tuner might start rotating. Zenith management loved the concepts proven by Polley's Flashmatic and directed his engineers to develop a better remote control.

First thoughts pointed to radio. But, because they travel through walls, radio waves could inadvertently control a TV set in an adjacent building or room. Using distinctive sound signals was discussed, but Zenith engineers believed people might not like hearing a certain sound that would become characteristic of operating the TV set through a remote control. It also would be difficult to find a sound that wouldn't accidentally be duplicated by either household noises or by the sound coming from TV programming.

Robert Adler's solution was for the remote to communicate with the TV by sound, not light, specifically, by ultrasound, that is, at frequencies higher than the human ear can perceive. Adler's remote control unit itself was very simple, and didn't even require any batteries.

The buttons struck one of four lightweight aluminium rods inside the unit, like a piano's keys strike its strings.

The receiver in the TV interpreted these high-frequency tones as signalling channel-up, channel-down, sound on/off, or power on/off. The necessary 30% increase in cost was imposing to consumers at first, but there was no doubt about the popularity of the system. The creator of the first practical wireless TV remote control, Dr. Robert Adler, paved the way for TV viewers to become couch potatoes more than 40 years ago.

In the 1960s, Adler modified his system to generate the ultrasonic signals electronically. Over the next twenty years, the ultrasound TV remote control was slowly becoming a standard adjunct to the television. By the time remote technology moved on to infrared light technology in the early 1980s, more than nine million TVs had been sold with Adler's remote control system.

Sony SIRC (Serial Infra Red Control) protocol.

With that small (but very relevant) history lesson over, we can now look at the method Sony use for infrared communications. The Sony SIRC protocol is both simple in it's design and elegant in its solution, and is a favourite of mine because of these two points. The protocol takes our earlier discussion of time as a third state, and proves its concept beyond a shadow of a doubt.

SIRC (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 infrared sensor's 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, and a 7-bit (0..127) button code, which represents the actual button pressed on the remote handset. The packet is transmitted most significant bit first (*MSB*), with the device code being sent first, then the button code (see right).
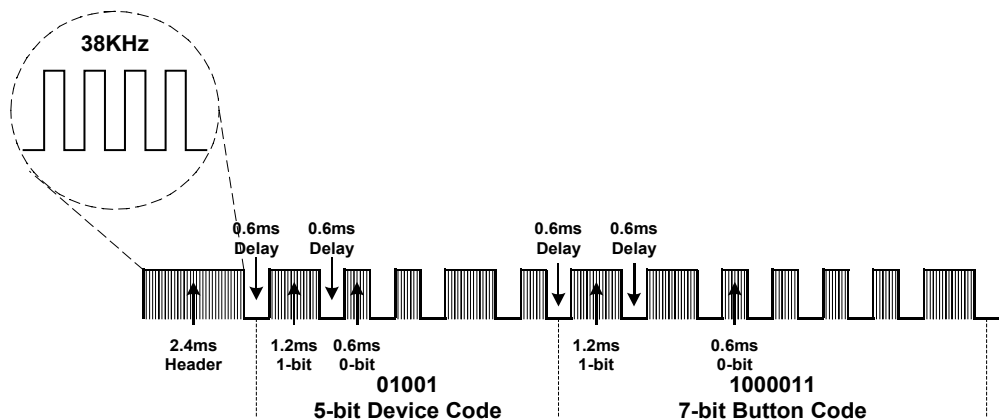
Examining the timing diagram, you should see a vague similarity to the Morse code timing diagram, in that different lengths of a state, mean different things. i.e. a DOT, or a DASH. The main difference being that the SIRC protocol is intended solely for a machine to interpret.

Sony SIRC Receiver.

We'll now take a look at a program that interprets the data stream transmitted from a Sony handset into ASCII numbers that we humans understand.

Load the program **SONY_REC.BAS** from the **IR_SAMPLES** folder, or type in the program from the listing shown below: -

```
' Sony Infrared receiver for the PROTON IR board

Device = 16F819
XTAL = 4
SERIAL_BAUD = 9600
RSOUT_PIN = PORTB.6
RSOUT_MODE = TRUE
RSOUT_PACE = 1


Dim HEADER       As Word                    ' Header pulse length
Dim PACKET       As HEADER                  ' 12-bit IR information
Dim P_VAL  As Byte                  ' The bit length
Dim IR_BUTTON    As Byte                    ' The BUTTON code
Dim IR_DEV       As Byte                    ' The DEVICE code
Dim SONY_LP      As Byte                    ' Loop variable

Symbol IR_SENSOR = PORTB.0                  ' Assign the IR Sensor
'----------------------------------------------------------------
Delayms 500                                 ' Wait for PICmicro to stabi-
lise           Input IR_SENSOR             ' Make the sensor pin an
input
Goto MAIN                                   ' Jump over the subroutine
'------[IR RECEIVE SUBROUTINE]-------------------------------------
' Receive a signal from a Sony remote control
' The button value is returned in IR_BUTTON
' The device is returned in IR_DEV
' IR_DEV will return holding 255 if an invalid header was received
IRIN:
Set IR_DEV
Set IR_BUTTON
If IR_SENSOR = 0 Then Return    ' Return if we're already inside a packet
HEADER = Pulsin IR_Sensor,Low      ' Receive the header
If HEADER < 200 OR HEADER > 270 Then Return ' Exit if invalid
Clear SONY_LP
Repeat                                      ' Implement a loop for the 12 bits
P_VAL = Pulsin IR_SENSOR,Low
Clear PACKET.11                             ' Default to a clear bit (zero-bit)
If P_VAL >= 90 Then Set PACKET.11           ' Should the bit be set ?
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
' Split the 7-bit BUTTON, and the 5-bit DEVICE code
IR_BUTTON = PACKET & %01111111                 ' Mask the BUTTON code bits
PACKET = PACKET << 1                           ' Move bit 7 into bit 8
IR_DEV = PACKET.Highbyte & %00011111   ' Mask the DEVICE code bits
Return
'------[MAIN PROGRAM LOOP]-------------------------------------------------
MAIN:
While 1 = 1                                    ' Create an infinite loop
Gosub IRIN                                     ' Receive an IR signal
If IR_DEV = 255 Then MAIN                ' Was the header valid ?
Rsout "DEVICE CODE = " , DEC IR_DEV , 13 ,_
"BUTTON CODE = " , DEC IR_BUTTON,13,13' Yes. So display the result
Wend                                           ' Do it forever
```

The program itself, although it looks complex, is very simple indeed. The actual receiving part is inside the subroutine named IRIN. This looks for a valid header of 2.4ms, by using the very handy **PULSIN** command. With a 4MHz crystal, the **PULSIN** command has a resolution of 10ms, so a pulse measuring 2.4ms (2400us) will return a value of 240. If the pulse is found to lie between 2000ms and 2700ms then it is assumed a valid header has been found, and the program creates a loop for the 12-bits of the packet to receive. Otherwise it exits prematurely with an error code.

Within the loop, each pulse is measured, again using the **PULSIN** command, and if it's found to be over 90ms (i.e. a value of 90) then it is assumed to be a set bit, if it's less that 90ms then it is assumed to be a clear bit. This is carried out for all 12-bits, then the 7-bit BUTTON and 5-bit DEVICE codes are split into their correct variables by using simple shifts and masks.

To test the code, you'll need a Sony remote control handset, or a Universal handset configured for Sony appliances. If your using a Universal handset, then configure it as a television remote. If you don't have a Sony, or Universal handset, then we'll look at building a compatible transmitter program soon.

Leave the PROTON IR board in its programming cradle after downloading the program, and open the serial terminal built into the compiler's IDE, by clicking on VIEW->SERIAL TERMINAL (see right).
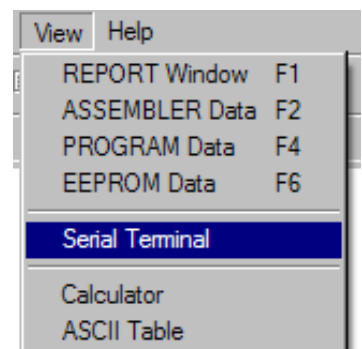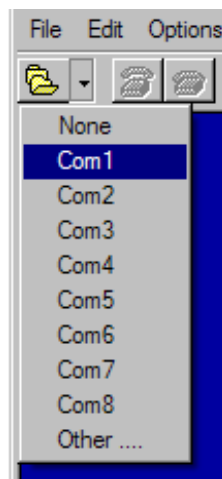
You should be greeted with a window looking something like the screenshot below: -
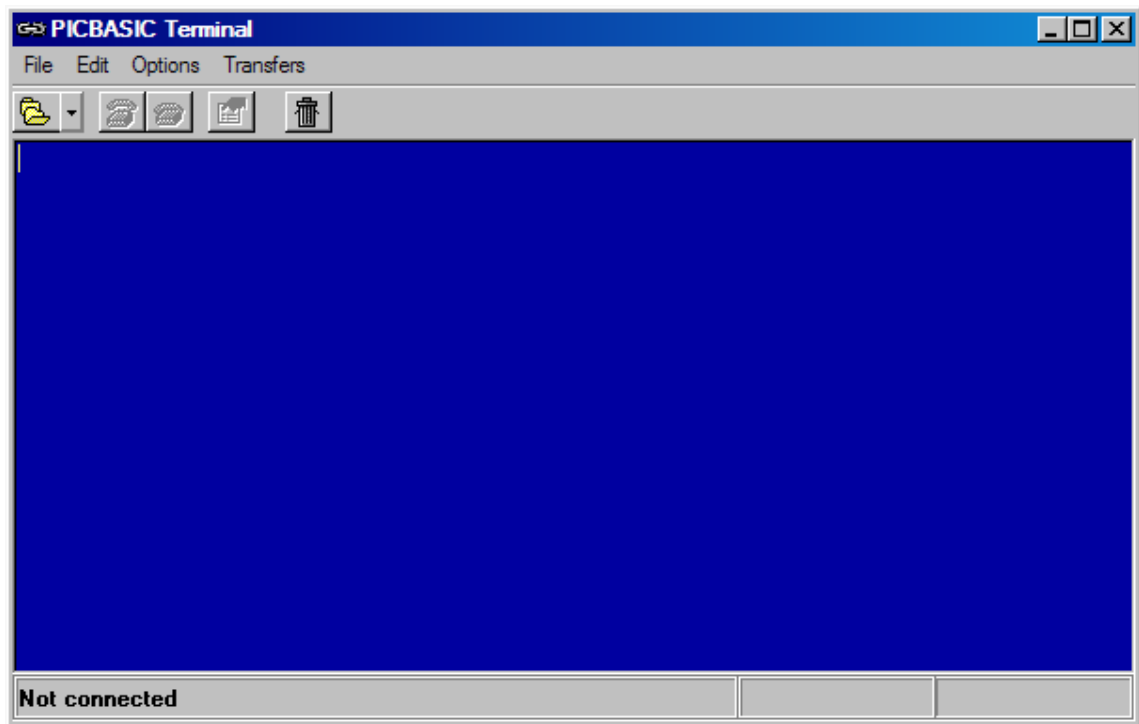
Now the Com port and Baud rate requires setting up. Click on the open Com icon, and a small menu will appear: -


Choose the appropriate Com port, according to the setup of you're PC. The illustration above shows Com1 being chosen. Note: that the Com port chosen should be the same as the Com port used to download the program to the PROTON IR. When the Com port is chosen, another window will appear that will allow the Baud rate to be set (see overleaf).
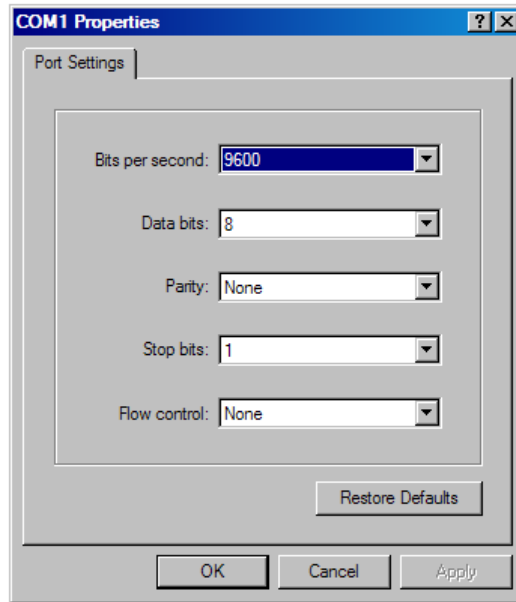
Set the Baud rate to 9600 (as above), and we're ready to receive data from the PROTON IR sitting in its cradle.

Point the handset in the general direction of the PROTON IR, and press the "2" button. You should see the screenshot below: -
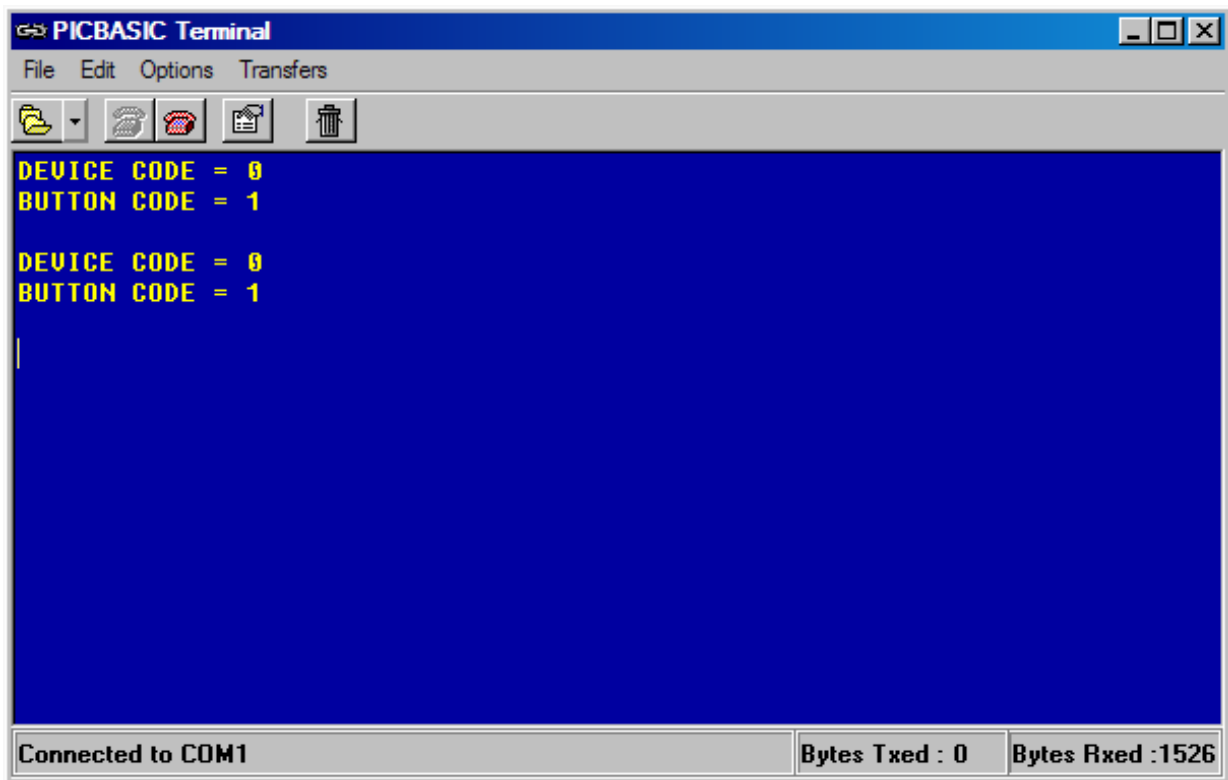
What the values displayed are telling us is that the device that the remote is talking to is a Satellite Receiver (a value of 0), and that the 2 button has been pressed.

However, these values are not set in stone, and can change from remote to remote, depending on the manufacturer. Shown below are the DEVICE, and BUTTON values that can be expected if an authentic Sony remote handset is used.

Code	Device
0	Satellite Receiver
1	Television receiver
2	VCR 1
4	VCR 2
6	Laser disk player
12	Surround sound unit
16	Cassette deck/tuner
17	CD player
18	Equaliser

DEVICE Codes for a Sony Handset.

Code	Function
0-9	Buttons 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

BUTTON codes for a Sony Handset.

Sony SIRC Transmitter.

As promised earlier, here's the code and explanation of a suitable transmitter for the previous Sony receiver example.

Transmitting data is inherently less complicated that receiving data, in that we simply send out a sequence of pulses at differing lengths for a given state. We do not need to capture and measure any incoming pulses. The full program of the transmitter is shown below: -

```
Include "IR_SETUP.INC"          ' Setup the modulation frequency
' Declare some variables
Dim IR_WORD as Word             ' Holds the 12-bit packet word to Transmit
Dim IR_BYTE as Byte             ' The button pressed, code (0..127)
Dim IR_CMD as Byte              ' Device code (0..31)
Dim S_LOOP as Byte       ' Temp variable us to build up the 12-bit
packet
Dim DEMO_LOOP as Byte     ' Demonstration loop variable
Goto OVER_SONY_TX               ' Jump over the subroutine
' Transmit a 12-bit packet using the Sony SIRC protocol.
' The Device code (0..31) is loaded into IR_CMD
' The Button data (0..127) is loaded into IR_BYTE
SONY_OUT:
IR_WORD = 0                     ' Clear IR_Word before we start
IR_WORD.highbyte = IR_Cmd ' Place the Device code in the top of IR_Word
IR_WORD = IR_Word >> 1          ' Move it down into the 7th bit position
IR_BYTE.7 = 0                   ' Make sure we can only send upto 127
IR_WORD = IR_WORD | IR_BYTE ' OR the IR_Byte value into IR_WORD
```

```
Pulsout SER_DATA,240,HIGH   ' Send the 2400us header pulse
Delayus 600                      ' Keep low for 600us
S_LOOP = 0
Repeat                           ' 12-bits to sent, least significant bit first
If Getbit IR_WORD,S_LOOP = 1 then     ' Check the individual bits of
IR_WORD
Pulsout SER_DATA,120,HIGH   ' If bit is 1 send pulse for 1200us
Else                             ' Otherwise…
Pulsout SER_DATA,60,HIGH    ' It's zero, so send pulse for 600us
Endif
Delayus 600                      ' Keep low for 600us
Inc S_LOOP
Until S_LOOP > 11                ' Close the loop
Delayms 34                       ' Make the total time up to approx 45ms
Return
'----------------------------------------------------------------
'  The main demonstration program loop starts here
OVER_SONY_TX:
IR_CMD = 15                              ' Set the Device code to 15
While 1 = 1
For DEMO_LOOP = 0 to 126                 ' Start counting from 0 to 126
IR_BYTE = DEMO_LOOP                      ' Place the loop value into IR_BYTE
Gosub SONY_OUT                          ' Transmit the IR signal
Delayms 300                             ' Delay between values sent
Next
Wend                                    ' Do it forever.
```

The above program named **SONY_TX.BAS** can be found in the PROTON+ compiler's directory, inside the **IR_SAMPLES** folder.

The Sony transmitting code hinges around a single subroutine named **SONY_OUT**. This takes the 5-bit DEVICE code pre-loaded into variable **IR_CMD**, and the 7-bit BUTTON code pre-loaded into variable **IR_BYTE**, and combines them both into variable **IR_WORD** which is then transmitted after the 2400us header pulse is sent.

Within the subroutine is a loop that accommodates all bits of the packet. The first 12-bits of variable **IR_WORD** are examined, and if the bit is found to be a 1, then a pulse of 1200us is initiated, if the bit is found to contain a 0, then a pulse of 600us is initiated. Each pulse, including the header has a 600us delay placed between it and the next one.
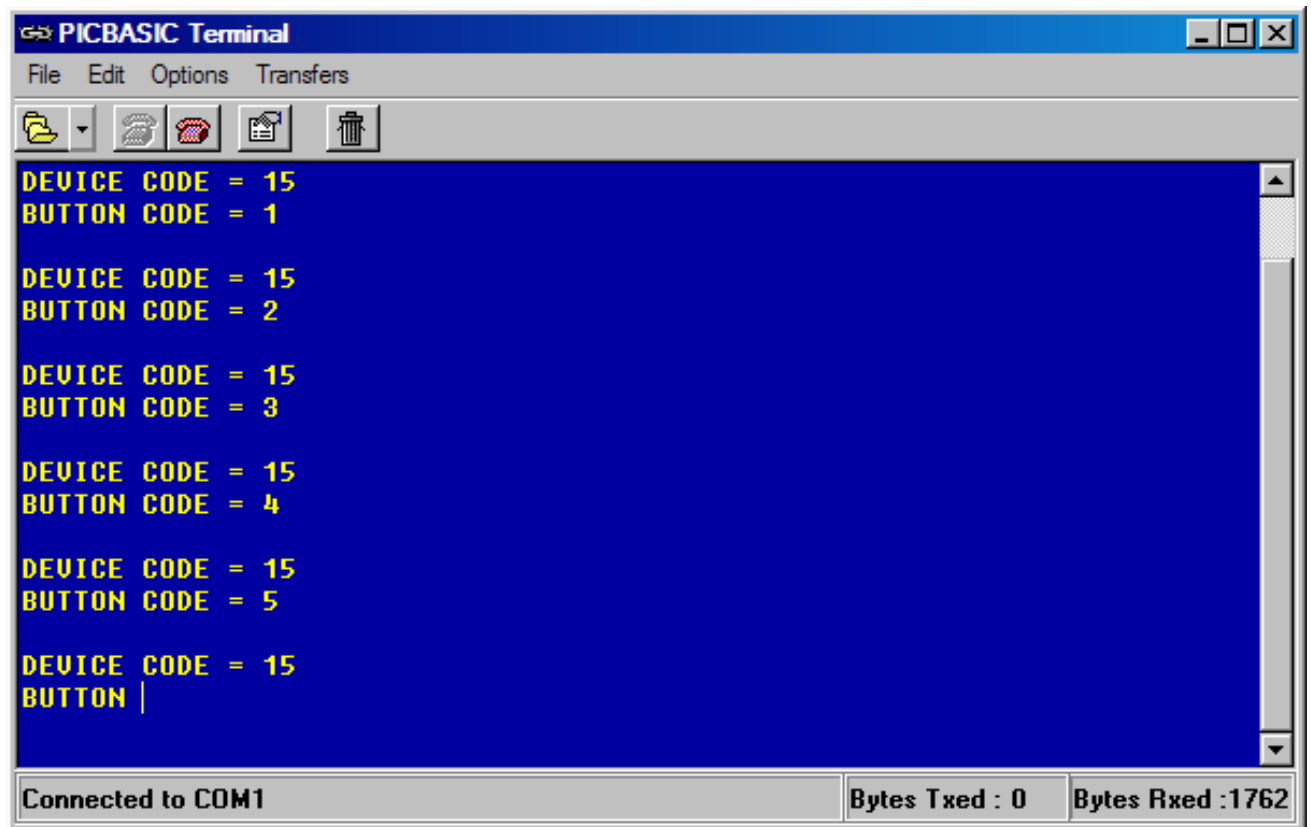
To demonstrate the SONY_OUT subroutine, a simple loop counting from 0 to 126 is implemented, each value of the loop is transmitted as the BUTTON code. The DEVICE code is preset to 15.

There are two methods of testing the Sony transmitter/receiver combination. You can either use two PROTON_IR boards, or if you own a PROTON development board you can use this as the Sony receiver.

Using two PROTON IR boards is a simple case of programming each one with the relevant code i.e. transmitter and receiver. Leave the receiver module in the programming cradle (as explained in the earlier Sony receiver discussion), and connect the transmitter module to a suitable power supply, noting polarity, and voltage. When both are powered up, you will see the serial terminal showing a count from 0 to 126 for BUTTON, and a fixed value of 15 for DEVICE (shown below).

To use the PROTON development board, load the program **PROTON_SONY_REC.BAS**, located in the compiler's directory inside the **IR_SAMPLES** folder. Connect jumper **J3** to enable the infrared sensor, and open the serial terminal program. As with the PROTON IR receiver code, you will see the terminal displaying a count from 0 to 126 for BUTTON, and a fixed value of 15 for DEVICE (shown below).

Phillips RC5 Protocol.

The RC5 protocol developed by Phillips, is arguably the most common type of commercially used protocol to date. This doesn't necessarily mean that it's a better method of data transmission than Sony's SIRC. Although saying that, the RC5 protocol does have a slightly higher reliability rating than the Sony SIRC. But this comes at a cost of more complexity. It could also be that Phillips produce dedicated transmitter/receiver ICs for their protocol, therefore manufacturers looking for an infrared medium have a ready made solution at hand.

Instead of different pulse widths simply signifying a 1 or a 0, as in SIRC, the RC5 protocol uses fixed pulse widths of 1778us and a bi-phase method, also known as Manchester encoding. In which a 1 is made up of a LOW to HIGH transition, and a 0 is a HIGH to LOW transition. A LOW to LOW, or HIGH to HIGH transition signifies an error in the data stream (shown below).

Bi-Phase coding diagram.

Looking at the diagram above shows that the bi-phase used by the RC5 protocol essentially has four states, 1, 0 and two invalids. Hence the more reliable data stream. Bi-phase is commonly used for data transmission because of this extension of states, and allows the trapping of invalid bits early in the bit stream. It also offers a more stable bit stream because a 1 or 0 state is more easily distinguishable.

The RC5 packet is a 14-bit word coded signal (see below). The first two bits are start bits, always having the value 1. The next bit is a control bit or toggle bit, which is inverted every time a button is pressed on the remote control transmitter. Five system bits hold the system (DEVICE) address so that only the right device responds to the code. Usually, Television receivers have the system address 0, VCRs the address 5 and so on. The command (BUTTON) sequence is six bits in length, allowing up to 64 different commands per address. Each bit length is 1.778ms, and the packet is repeated approximately every 114 ms.
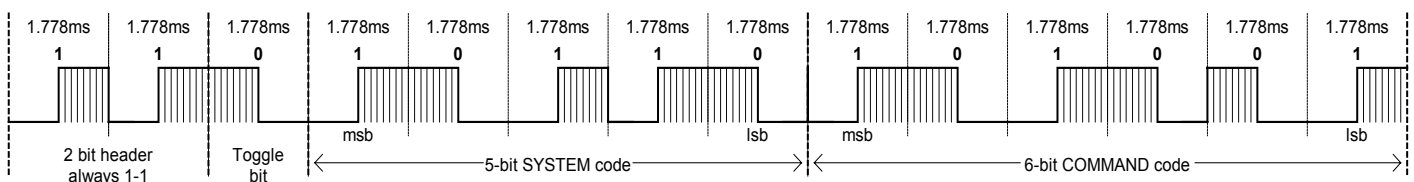
RC5 Timing.

RC5 Receiver.

Because of the bi-phase method that the RC5 protocol uses, the receiving code is more involved than it's Sony counterpart. And is shown in full below:

```
' Decode an RC5 datastream

Device = 16F819
XTAL = 4

SERIAL_BAUD = 9600
RSOUT_PIN = PORTB.6
RSOUT_MODE = TRUE
RSOUT_PACE = 1

Dim RC5_SERIAL_BUF as Dword      ' Buffer for incoming bitstream
Dim RC5_TOGGLE as Byte              ' The RC5 Toggle bit
Dim RC5_SYSTEM as Byte              ' The RC5 SYSTEM byte
Dim RC5_COMMAND as Byte             ' The RC5 Command byte
Dim RC5_BIT_COUNT as Byte        ' Counter for incoming bits
Dim RC5_TEMP_BYTE as Byte        ' Holds the 2-bit pattern
Dim RC5_PARSE_FLAG as Byte       ' PHASE_DECODE returns data in this
Dim RC5_RAW_DATA as Word            ' Holds the RAW decoded bits

Symbol RC5_DATA_IN = PORTB.0      ' Input data from IR pickup
Symbol IR_VALID = RC5_PARSE_FLAG.7
Symbol ONE = %01000000              ' 2-bit incoming pattern matches
Symbol ZERO = %10000000

'-------------------------------------------------------------------

Delayms 500                        ' Wait for PICmicro to stabilise
ALL_DIGITAL = TRUE
Goto OVER_RC5_SUBS                  ' Jump over the subroutines

'-----[SUBROUTINES]-------------------------------------------------
' Enter with two-bit data in RC5_TEMP_BYTE
' return with result code in RC5_PARSE_FLAG.
' RC5_PARSE_FLAG.0 = valid data, one or zero
' RC5_PARSE_FLAG.1 = unused
' RC5_PARSE_FLAG.2 = set if data is invalid

PHASE_DECODE:
Clear RC5_PARSE_FLAG             ' Clear PARSE FLAG before we start
If RC5_TEMP_BYTE = ONE Then         ' Compare to bit pattern '01'
Set  RC5_PARSE_FLAG.0            ' Return with valid data in LSB
Return
Endif
If RC5_TEMP_BYTE = ZERO Then Return ' Compare to bit pattern '10'
Set RC5_PARSE_FLAG.2             ' Bit pattern neither 01 nor 10
Return
```

```
'-----------------------------------------------------------------
' Read and decode the RC5 input data stream
' Returns prematurely with IR_VALID set if there was a problem
' RC5_SYSTEM returns the 5-bit SYSTEM byte
' RC5_COMMAND returns the 6-bit COMMAND byte
' RC5_TOGGLE returns the TOGGLE bit
READ_RC5:
Clear RC5_PARSE_FLAG              ' Clear PARSE FLAG before we start
While RC5_DATA_IN = 1 : Wend          ' Wait for an IR signal
Delayus 440                           ' Wait for 440us. (1/4 bit time)
If RC5_DATA_IN = 1 Then               ' Is IR signal logic 0 ?
Set IR_VALID                          ' Yes.. So set IR_VALID to show er-
ror
Return                                ' And exit prematurely
Endif
' Detecting the IR signal (above), also ate the first START bit
' So.. capture the remaining 26 phases of the RC5 packet
Clear RC5_BIT_COUNT
Repeat                                ' Create a loop for remaining phases
Delayus 880                           ' Wait for 880us. (1/2 bit time)
Set RC5_SERIAL_BUF.0          ' Default to bit set
If RC5_DATA_IN = 1 Then Clear RC5_SERIAL_BUF.0 ' Test IR data stream
RC5_SERIAL_BUF = RC5_SERIAL_BUF << 1
Inc RC5_BIT_COUNT                     ' Increment the bit counter
Until RC5_BIT_COUNT = 31              ' Loop back and fill all 32 bits
' Decode the saved data stream into the various RC5 bytes and flag.
RC5_BIT_COUNT = 0
Repeat
RC5_TEMP_BYTE = RC5_SERIAL_BUF.Byte3 & %11000000
RC5_SERIAL_BUF = RC5_SERIAL_BUF << 2
Gosub PHASE_DECODE
RC5_RAW_DATA = RC5_RAW_DATA << 1 ' Shift up 1 bit position
If RC5_PARSE_FLAG.2 = 1 Then          ' Illegal pattern found ?
Set IR_VALID                          ' Set IR_VALID to show error
Return                                ' And exit prematurely
Endif
RC5_RAW_DATA.0 = RC5_PARSE_FLAG.0     ' Move bit into RC5_RAW_DATA
Inc RC5_BIT_COUNT
Until RC5_BIT_COUNT > 12              ' Have we done all 13 bits?
RC5_TOGGLE = RC5_RAW_DATA.11          ' Extract the toggle bit
RC5_COMMAND = RC5_RAW_DATA.Lowbyte & %00111111 ' Extract the COMMAND byte
RC5_RAW_DATA = RC5_RAW_DATA << 2 ' Shift the raw data up two bits
RC5_SYSTEM = RC5_RAW_DATA.Highbyte & %00011111 ' Extract the SYSTEM byte
Clear IR_VALID                        ' Clear IR_VALID to show a good re-
turn
Return
'-----[MAIN PROGRAM LOOP STARTS HERE]---------------------------------
OVER_RC5_SUBS:
While 1 = 1
Gosub READ_RC5                        ' Decode the RC5 data (if any!)
If IR_VALID = 1 Then OVER_RC5_SUBS    ' If a problem occurred, look again
Rsout "SYSTEM " , DEC RC5_SYSTEM , 13
Rsout "COMMAND " , DEC RC5_COMMAND , 13
Rsout "TOGGLE " , DEC RC5_TOGGLE , 13
Wend
```

The program overleaf named **RC5_REC.BAS**, can be found in the **IR_SAMPLES** folder, located in the compiler's directory. Don't be deterred by the relative complexity of the BASIC code, its action is quite simple, if not a bit long winded. Because of the extra complexity involved, we'll take a closer look at its operation.

Collecting data.

The program hinges around the main subroutine named **READ_RC5**. This waits in a tight loop monitoring the infrared sensor's output. When a logic low is detected (an IR signal), the subroutine waits for approx 444us which is a quarter of the actual bit length of 1.778ms (*1778us / 4 = 444.5us*) see the RC5 timing diagram shown earlier. This will place all future bit sampling at the centre of the phases (see bi-phase discussed earlier). A test is again made of the infrared sensor's output, and if found NOT to be a logic low, the error flag **IR_VALID** is set, and the subroutine exits prematurely. If the sensor is found to be producing a logic low, then we have effectively detected what could be the first of the two start bits.

A loop is then created to encompass, detect, and store, the remaining 26 phases of the 14-bit packet. Within the loop, a delay of 888us is implemented before examining the infrared sensor's output. This will place the sampling to near the centre of each phase. The diagram below should help illustrate this method more clearly: -
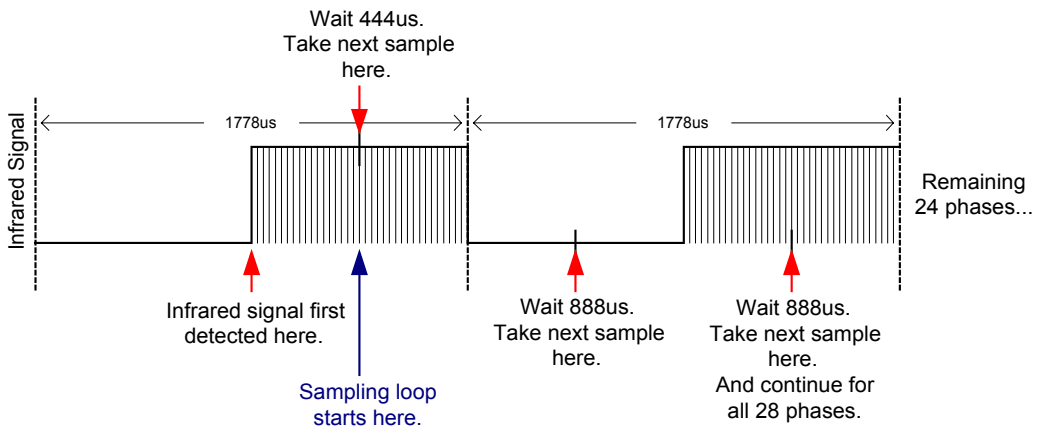
Each
phase's state is placed into a 4 byte buffer named **RC5_SERIAL_BUF** as bits, with a high phase being stored as a set bit, and a low phase being a clear bit. This is illustrated in the diagram below.
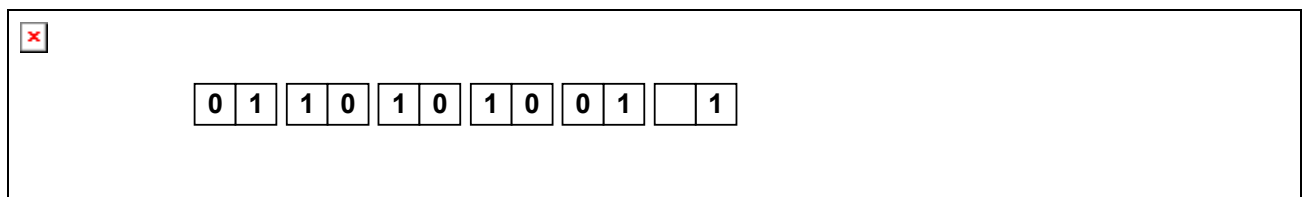
In essence, we have now actually captured an RC5 packet, and could work with the values placed in the 4 byte buffer. However, we humans like some order in our results, so we'll parse the buffer into its separate pieces. i.e. TOGGLE bit, SYSTEM byte, and COMMAND byte. In the process of parsing, we can also detect any inconsistent results (see Bi-Phase coding diagram) and act upon them accordingly.

Order out of Chaos.

Another loop is created in order to extract all 26 phases from the buffer. Within the loop, the last byte (byte 3) of buffer **RC5_SERIAL_BUF** is loaded into temporary variable **RC5_TEMP_BYTE**, and masked so that only the relevant last two bits can be seen: -

```
RC5_TEMP_BYTE = RC5_SERIAL_BUF.Byte3 & %11000000
```

Variable **RC5_SERIAL_BUF** is then shifted left two positions, in order to discard the two bits just extracted; and move the next two into position in preparation for the next time around the loop. If we take the values presented in the previous diagram of the contents of **RC5_SERIAL_BUF**, variable **RC5_TEMP_BYTE** contains the binary value 01000000 (i.e. the last two bits of **RC5_SERIAL_BUF** masked).

We now need a method of decoding the bi-phase (Manchester) information into a valid bit value. This is performed by the subroutine **PHASE_DECODE**, which examines the contents of **RC5_TEMP_BYTE** and compares it with the known values for a 1 or a 0 (ONE is binary 01000000, and ZERO is binary 10000000). If a decoded value of 1 is found, then bit 0 of variable **RC5_PARSE_FLAGS** is set, and if it's found to be 0, then this bit is cleared. If neither a 1 nor a 0 is found, then bit 2 of **RC5_PARSE_FLAGS** is set to indicate an invalid phase pair. This is the strength of bi-phase coding schemes.

Upon the return of **PHASE_DECODE**, a test is made of **RC5_PARSE_FLAGS.2**, if it's found to be set, then the loop and subroutine are exited prematurely. If however, a valid phase was found, then the contents of **RC5_PARSE_FLAGS.0** is loaded into bit 0 of variable **RC5_RAW_DATA**. Upon each loop event, this variable is shifted left 1 place, thus building up the bits of the RC5 packet.

```
RC5_RAW_DATA.0 = RC5_PARSE_FLAG.0      ' Move bit into RC5_RAW_DATA
```

Once the loop has finished, extracting the two bytes and flag from the raw data is a simple process of masks and shifts, which is basically the same method used in the Sony SIRC decoder.

To extract the TOGGLE bit we simply place bit 11 of **RC5_RAW_DATA** into it's returning variable **RC5_TOGGLE**: -

```
RC5_TOGGLE = RC5_RAW_DATA.11                    ' Extract the toggle bit
```

Extracting the 6-bit COMMAND byte involves masking the bits we require using the bitwise AND operator (&): -

```
RC5_COMMAND = RC5_RAW_DATA.Lowbyte & %00111111 ' Extract the COMMAND byte
```

We're now left with two bits of the 5-bit SYSTEM byte occupying the lower half of the variable **RC5_RAW_DATA**, and three occupying the upper byte. Therefore we shift the whole variable left by 2 positions, and mask the SYSTEM bits from the upper byte: -

```
RC5_RAW_DATA = RC5_RAW_DATA << 2        ' Shift the raw data up two bits
RC5_SYSTEM = RC5_RAW_DATA.Highbyte & %00011111 ' Extract the SYSTEM byte
```

There you have it! an RC5 decoder program ready for testing. Follow the instructions explained in the Sony SIRC decoder discussion, but use an RC5 handset, or a universal type set for Phillips mode. You should see the terminal window produce the text shown below when button 1 is pressed on the remote handset: -
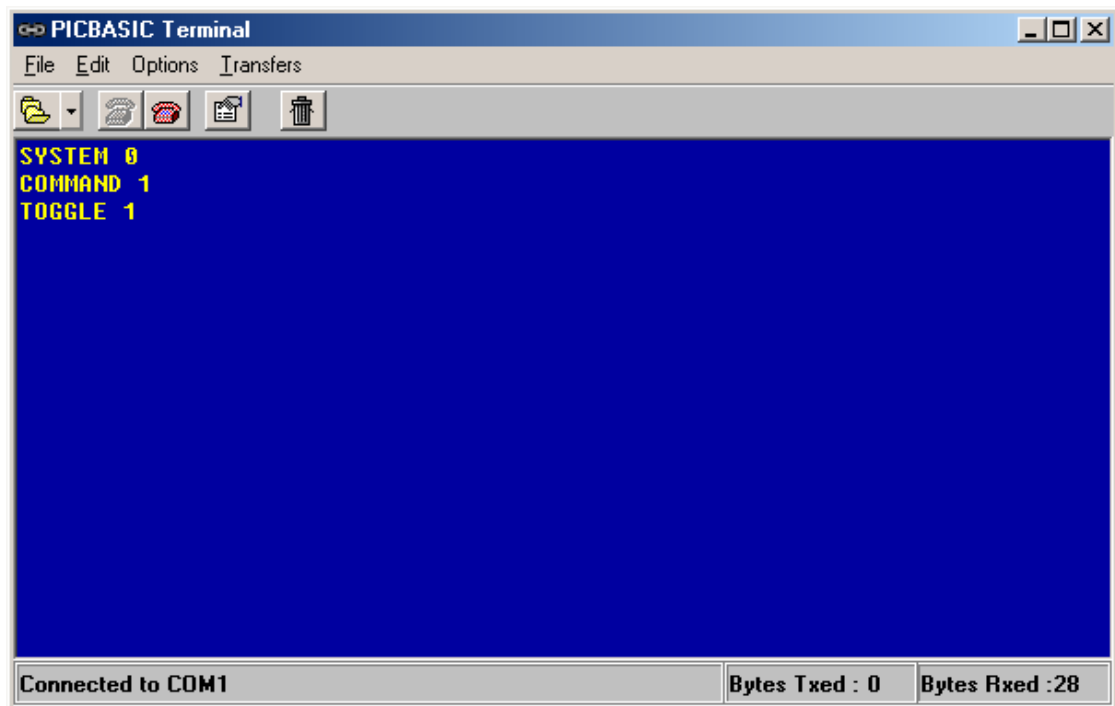
As with the Sony protocol, the SYSTEM and COMMAND values are not set in stone, and may change from manufacturer to manufacturer. Therefore don't be surprised if you see different values, as long as they are consistent.

As expected, you can use the PROTON development board for receiving the RC5 signal by loading the program **PROTON_RC5_REC.BAS**, located in the compiler's directory inside the **IR_SAMPLES** folder. Connect jumper J3 to enable the infrared sensor, download the code, and open the serial terminal program.

RC5 Transmitter.

What's that you say... You don't have a handset capable of producing RC5 signals?. Then read on, because there's one listed below: -

The RC5 transmitter program named **RC5_TX.BAS**, can also be found in the usual **IR_SAMPLES** folder.

```
' RC5 Remote Control Transmitter
' For use with the Crownhill PROTON IR

Include "IR_SETUP.INC"                 ' Setup the modulation frequency

Dim RC5_SYSTEM as Byte
Dim RC5_COMMAND as Byte
Dim RC5_BIT_NR as Byte
Dim RC5_TOGGLE as Byte

RC5_TOGGLE = 0                         ' Set initial TOGGLE state low
Low SER_DATA                          ' Set the SERIAL DATA pin to LOW
Goto OVER_RC5_TX               ' Jump over the subroutines
'-------------------------------------------------------------------
' Send a two-phase pulse for a LOGIC 0
' HIGH - LOW
LOGIC_0:
Set SER_DATA
Delayus 887
Clear SER_DATA
Delayus 884
Return
'-------------------------------------------------------------------
' Send a two-phase pulse for a LOGIC 1
' LOW - HIGH
LOGIC_1:
Clear SER_DATA
Delayus 887
Set SER_DATA
Delayus 884
Return
'-------------------------------------------------------------------
' Transmit SYSTEM, COMMAND, and TOGGLE as 14 bit RC5 code
RC5_XMIT:
RC5_TOGGLE = ~RC5_TOGGLE                ' Load TOGGLE bit
Gosub LOGIC_1                       ' Send first START bit
Gosub LOGIC_1                       ' Send second START bit
```

```
' Send TOGGLE bit
If RC5_TOGGLE = 0 Then Gosub LOGIC_0 : Else : Gosub LOGIC_1

RC5_BIT_NR = 5
Repeat                                  ' Send 5 bit SYSTEM, msb first
Dec RC5_BIT_NR
If Getbit RC5_SYSTEM,RC5_BIT_NR = 0 Then
Gosub LOGIC_0
Else
Gosub LOGIC_1
Endif
Until RC5_BIT_NR = 0

RC5_BIT_NR = 6
Repeat                                  ' Send 6 bit COMMAND, msb first
Dec RC5_BIT_NR
If Getbit RC5_COMMAND,RC5_BIT_NR = 0 Then
Gosub LOGIC_0
Else
Gosub LOGIC_1
Endif
Until RC5_BIT_NR = 0

Clear SER_DATA
Delayus 87                              ' Frame gap delay
Return
'------------------------------------------------------------------
' Main program loop starts here
OVER_RC5_TX:
While 1 = 1                             ' Create an infinite loop
RC5_SYSTEM = 1                          ' Set up SYSTEM byte as value 1
For RC5_COMMAND = 0 to 63      ' Create a loop for all COMMAND values
Gosub RC5_XMIT                          ' Transmit the data
Delayms 400                             ' Pause between transmissions
Next
Wend
```

As you can see, the transmitting program is a whole lot simpler than its receiver.

The heart of the program is located in subroutine **RC5_XMIT**, where variables **RC5_SYSTEM** and **RC5_COMMAND** are transmitted. Each of the BYTE sized variables is scanned using the very useful **GETBIT** command, and if a bit is set, then a call to **LOGIC_1** is performed, while a call to **LOGIC_0** is performed upon a clear bit being found.

The **TOGGLE** bit is simply inverted every time a call is made to **RC5_XMIT**, and transmitted prior to the SYSTEM and COMMAND bytes.

The modulation is then turned off for 87ms to signify a FRAME gap.

Each phase transmission is performed by subroutines **LOGIC_0** and **LOGIC_1**. **LOGIC_0** turns on the modulation for 887ms, then turns off the modulation for 884ms. **LOGIC_1** turns off the modulation for 887ms then turns on the modulation for 884ms. The differences in the delays is to counteract the time taken for the BASIC command to perform.

The main program loop simply pre-loads **RC5_SYSTEM** with a value of 1, and increments **RC5_COMMAND** within a **FOR-NEXT** loop.

Standard Serial Data.

Because the 38KHz modulation of the infrared LEDs is carried out as a background task and takes no cycles away from the main program, RS232 serial data can be transmitted and received via infrared simply by using the compiler's built in serial commands such as **RSIN**/**RSOUT** or **SERIN**/**SEROUT**. However, because of a settling time inherent within the infrared receiver module, transmission speeds of 300 baud (bits per second) should be classed as a maximum for reliable operations. Higher speed may be achievable, but reliability falls off sharply over distances of more than a few feet.

Programs utilising standard serial are extremely easy to write, and both the transmitter and receiver code will fit on this single page. They are shown below.

Transmit: -
```
' Transmit standard 300 baud serial data via infrared
Include "IR_SETUP.INC"              ' Set up the PROTON IR board
Dim COUNT_VAR as Byte
AGAIN:
For COUNT_VAR = 0 to 255            ' Create a loop of 0 to 255
Rsout Cls , "HELLO WORLD ", DEC3 COUNT_VAR, 13 ' Send data via IR
Delayms 400                        ' A small delay
Next                               ' Close the loop
Goto AGAIN                         ' Repeat forever
```

Receive with a PROTON IR: -
```
' Receive Standard asynchronous RS232 serial data
Include "IR_SETUP.INC"              ' Set up the PROTON IR board
While 1 = 1: Rsout Rsin : Wend      ' Display the data
```

Receive on a PROTON Development board: -
```
' Receive Standard asynchronous RS232 serial data
' On the PROTON development board.
Device = 16F877
XTAL = 4
LCD_DTPIN = PORTD.4
LCD_RSPIN = PORTE.0
LCD_ENPIN = PORTE.1
LCD_INTERFACE = 4                  ' 4-bit Interface
LCD_LINES = 2
SERIAL_BAUD = 300                  ' Slow baud rate for IR coms
RSIN_PIN = PORTC.0                 ' Receive via the IR receiver
RSIN_MODE = TRUE                   ' Set serial mode to TRUE
Delayms 500
ALL_DIGITAL = TRUE                 ' PORTA/PORTD to all digital
Dim BYTE_IN as Byte
Cls                                ' Clear the LCD
While 1 = 1: Print Rsin : Wend ' Display the data
```

How does the Programming Cradle work ?

Using the bootloader for programming the PROTON IR board is both quick and effective, offering a rapid turnaround for software development, however, it does not allow the configuration fuses to be set, which poses a problem if you wish to keep your latest masterpiece away from prying eyes.

Along with offering a serial interface, the Programming Cradle also has facilities for in circuit serial programming (ICSP), which affords the ability to place a program permanently, and safely inside the PROTON IR's on-board PICmicro^{tm}. But because this is a FLASH device, it can be erased and re-used as many as 100,000 times.

The Programming cradle is split into three sections. Power supply, Serial interface, and ICSP header. Each of these sections is shown below.

Power Supply.

The power supply is a conventional rectifier – regulator type, supplying 5 Volts to the board with approx 500mA to 750mA of current. Using the bridge rectifier allows both an external AC or DC supply of between 6 and 9 Volts to be used. Any higher than 9 Volts may cause the 7805 regulator to become unreliably hot.
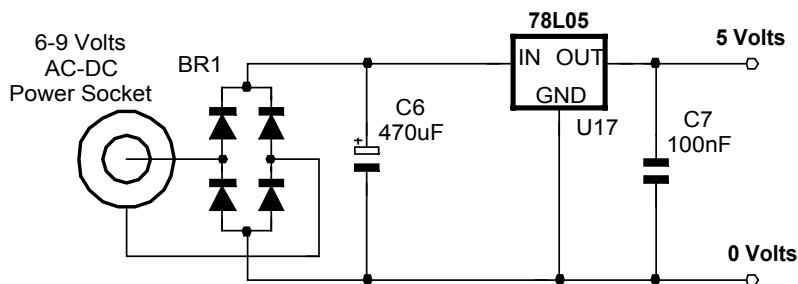
**Power Supply circuit.**

RS232 Transceiver.

The RS232 Transceiver section, uses the ever popular MAX232 device from MAXIM semiconductors. Serial communication is a useful aid to debugging microcontroller code. The MAX232 ensures that the correct voltage levels are seen by both the computer and the microcontroller, as well as serving to isolate the two devices somewhat.
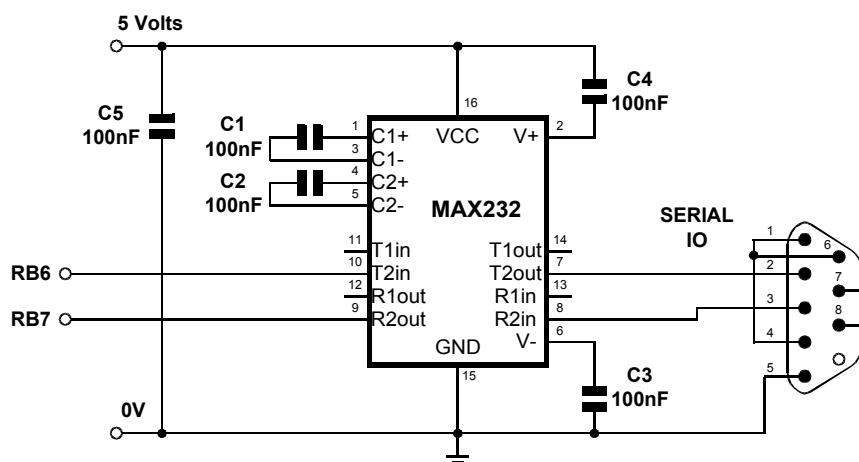
In-Circuit-Programming Header.

The In-Circuit-Programming section, allows the on-board 16F819 PICmicro[tm], to be programmed in circuit. A suitable programmer must be used, such as the microEngineering Labs EPIC[tm] programmer. The circuit for this section is shown below: -

The PICmicro[tm] programmer must be fitted to the 10-pin header, preferably using a short ribbon cable and suitable header sockets. The programming pins are connected to the relevant programming pins of the PICmicro[tm]. The inclusion of the push switch allows the PICmicro[tm] to be RESET while developing software. The 1N4148 diode stops any high voltage leaking into the 5 Volt supply when the microcontroller is being programmed, as most devices require at least 12 Volts on their VPP pin. The 1KΩ resistor stops the 5 Volt line from becoming short circuit when the RESET switch is operated.
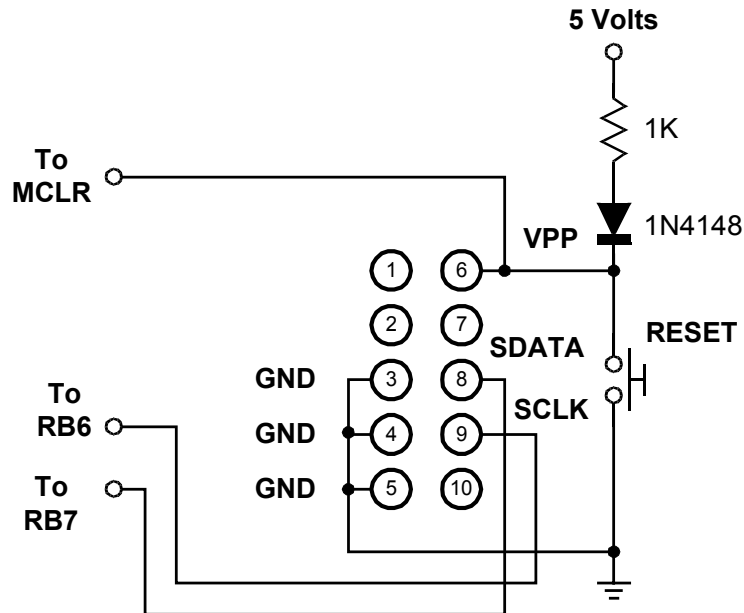
Connecting the EPIC programmer to the Cradle involves making a twist in the 10-way ribbon cable, so that the red stripe is pointing away from the LED on the EPIC board, and the red stripe is pointing towards the serial socket on the programming cradle.

Programming Cradle
Header

EPIC Programming
Header
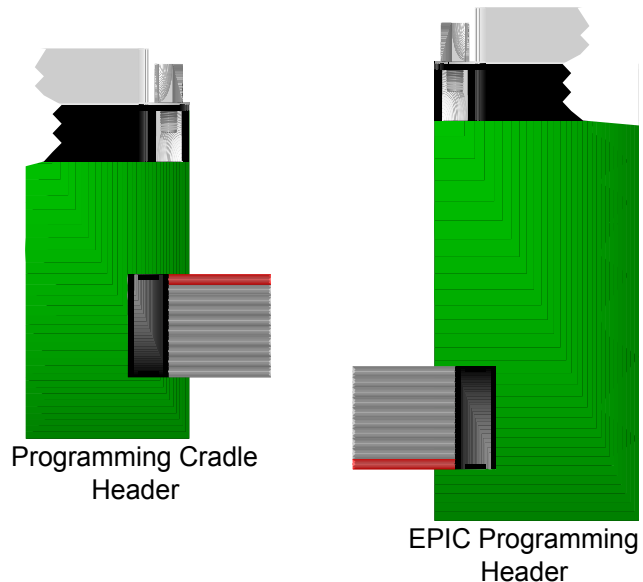
I hope I have succeeded in showing that infrared data communications is both fun and useful. And that it is not the black art that most people think.

Above all, I hope I have sparked your imagination, and that you have fun creating your own applications.

Les Johnson.