

The background is a bright yellow color, overlaid with a collage of various PIC microcontrollers. The chips are scattered across the page, some showing their top surfaces with labels like 'PIC16C62-04/P', 'PIC16C62-02/P', and 'PIC16C62-01/P'. Others show their bottom surfaces with pin connections. The text is centered and rendered in a bold, stylized font.

THE L.E.T PICBASIC COMPILER

USER MANUAL



Crownhill Associates

L.E.T PIC BASIC COMPILER

BASIC compiler for the : -

12C508, 12C509, 16C54, 16C55, 16C56, 16C57
16C71, 16F83, 16F84, 16F87x

range of PIC micro's

Please Note.

Although every precaution has been taken with the preparation of this book to ensure that any projects, designs or programs enclosed, operate in a correct and safe manner. The author and publisher assume no responsibility for errors or omissions. Neither is any liability assumed for the failure of any project, design or program, or any damage caused to equipment that it may be connected to, or used in combination with.

Copyright Crownhill Associates. All right reserved. No part of this publication may be reproduced, stored in a retrieval system, or distributed in any form or by any means without the written permission of the publisher or author.

The Microchip logo and name are registered trademarks of Microchip Technologies Inc.

The L.E.T PIC BASIC Lite, Pro, and Plus are registered trademarks of Crownhill Associates.

Table of Contents

1	Introduction	4
1.1	LET versus the rest	4
1.2	PIC Devices	4
1.3	Packages	4
1.4	LET Pic BASIC Discussion	5
1.5	Distributing Pic BASIC LITE	5
1.6	Distributing Pic BASIC Pro and Plus	5
1.7	Contact Details	5
2	Starting Out	6
2.1	Installing the software	6
2.2	OK what now?	7
2.3	What do all the other menu options do?	8
3	Program Rules	9
3.1	Program order	9
3.2	Labels	9
3.3	General statements	9
3.4	General hints on programming	10
4	Mathematical Operators and Comparators	11
4.1	Mathematical Operators	11
4.2	Comparators	12
5	L.E.T Pic BASIC commands	13
5.1	ADIN	14
5.2	ASM	15
5.3	BSTART	16
5.4	BSTOP	17
5.5	BUSIN	18
5.6	BUSOUT	19
5.7	BUTTON	20
5.8	CLEAR (or LOW)	21
5.9	CLS	22
5.10	COUNTER	23
5.11	CURSOR	24
5.12	DATA	25
5.13	DEFINE	26
5.14	DELAYMS	27
5.15	DELAYUS	28
5.16	DEVICE	29
5.17	DIM	30
5.18	EEDATA	31
5.19	END	32
5.20	FOR....TO....NEXT	33
5.21	GOSUB....RETURN	34

Table of Contents (continued)

5.22	GOTO	35
5.23	IF....THEN	36
5.24	INCLUDE	37
5.25	INIT	38
5.26	INKEY	39
5.27	INPORTA, INPORTB, INPORTC	40
5.28	[LET]	41
5.29	MEMREAD	42
5.30	MEMWRITE	43
5.31	OUTA, OUTB, OUTC	44
5.32	PEEK	45
5.33	POKE	46
5.34	PRINT	47
5.35	READ	48
5.36	REM	49
5.37	RESTORE	50
5.38	RSIN	51
5.39	RSOUT	52
5.40	SET (or HIGH)	53
5.41	SOUND	54
5.42	SLEEP	55
5.43	STOP	56
5.44	STORE	57
5.45	SWAP	58
5.46	SYMBOL	59
5.47	TIMER	60
6	Using the Plus version of the compiler	61
6.1	Page boundaries	61
6.2	Analogue pin issues	61
6.3	Using the ADIN command	61
7	The on-board Programmer	62
7.1	Using the on-board Programmer	62
8	Universal PIC Programmer	63
8.1	Using the Programmer	63
8.2	18 pin Jumper settings	64
8.3	28 pin Jumper settings	64
9	Adapter Layouts	65
9.1	8 pin Adapter layout	65
9.2	28 pin Adapter layout	66
9.3	40 pin Adapter layout	67
10	Overview of the programming software	68
10.1	What do all the other menu options do ?	69
10.2	Learning by doing	70

1 - Introduction

The LET Pic BASIC compilers were written with simplicity in mind. Using BASIC, which is probably the easiest programming language around, you can now produce quite intricate applications for your PIC without having to learn the ins and outs of assembler. Unlike other 'BASIC' compilers around, many of which bear little resemblance to real BASIC, the authors of this version have tried to keep as much to the original ideals of BASIC as possible. Having said this, they have included various 'enhancements' for extra versatility and ease of use.

1.1 - LET versus the rest

LET Pic BASIC provides a seamless development environment, found with no other Pic BASIC. With LET Pic BASIC, you write, debug and compile your code within the same Windows application, and by using a compatible programmer, just one key press allows you to brogan and verify the resulting code in the PIC of your choice!

The LET front end is fully Windows based. Simply specify the device at the program beginning and the code produced will be fully compatible with that device.

It has also been noted that most compilers are incapable of producing code for the PIC16C5x range, which of course LET Pic BASIC does. It should be noted that because LET Pic BASIC, is as close to a true BASIC as possible, it is NOT code compatible with the popular Parallax PICBASIC which is a proprietary language, specific to their BASIC Stamp Parts.

1.2 - PIC Devices

The devices supported by this software are the most commonly used and the LET Pic BASIC takes advantage of their various features e.g. The A/D converter in the 16C71, the data memory eeprom area in the 16C84 and 16F84. This manual is not intended to give you details about PIC devices So for further information visit the Microchip website at www.microchip.com, and download the various datasheets available. Of course, if you have purchased the LET Pic BASIC Pro you will already have the data sheets on your CD. Let's not forget Pic BASIC Plus, for those of you who want to take advantage of the incredibly cost effective 16F87x series of micro's.

1.3 - Packages

Those of you who have purchased the PRO or PLUS version can take advantage of the predefined packages which can be included in your programs to access things such as LCD, Keypads, I²C Bus and others. This makes it incredibly easy to get a powerful application incorporating input and output up and running quickly.

1.4 - LET Pic Basic Discussion

For your convenience we have set up a web site **www.letbasic.com**, where there is a section for users of Pic BASIC to discuss the compiler, and provide self help with programs written for LET Pic BASIC, or download sample programs. The web site is well worth a visit now and then either to learn a bit about how other peoples code works or to request help should you encounter any problems with programs that you have written.

1.5 - Distributing Pic BASIC LITE

Please feel free to pass the Pic BASIC LITE around, we ask only that you include this documentation or at least a link to our site where the file can be downloaded so we know people are getting the most out of the software. Please DO NOT alter the files in any way.

1.6 - Distributing Pic Basic PRO and PLUS

Quite simply - DON'T ! Pic BASIC Pro and Pic BASIC Plus are covered by copyright and any unauthorised distribution, loan, selling, or copying is prohibited by law.

1.7 - Contact Details

Should you need to get in touch with us for any reason our details are as follows: -

Postal: Crownhill Associates Limited
32 Broad Street
Ely, Cambridgeshire
CB4 4AH

Telephone: UK: 01353 666709
Int: +44 1353 666709

Fax: UK: 01353 666710
Int: +44 1353 666710

Email: **Sales@crownhill.co.uk**
Web Site: **http://www.crownhill.co.uk**
http://www/letbasic.com

2 - Starting Out

2.1 - Installing the software

First things first- if you haven't already set the software up, you will have to use one of two methods depending on how you received the software.

Method 1 - If you downloaded the file from the internet

Locate the file you just downloaded on your machine and double-click it. You will need WinZip or some other extraction utility that can handle ZIP files to retrieve the information. In most cases you will be prompted as to where you would like to extract the files to. Bear in mind that this is only temporary as you will only be extracting the **setup** files and not the working program itself.

Once you have extracted the files you will see a program called **setup** or **setup.exe** this is the main install application. Double-click this and follow the on-screen prompts.

Method 2 - If you received the files on CD from us

Using Windows explorer, change to your CD and locate the directory labelled Pic Basic Lite or Pic Basic Pro depending on which version you have. Double-click this directory and then locate the program called **setup** or **setup.exe** this is the main install application. Double-click this and follow the on-screen prompts.

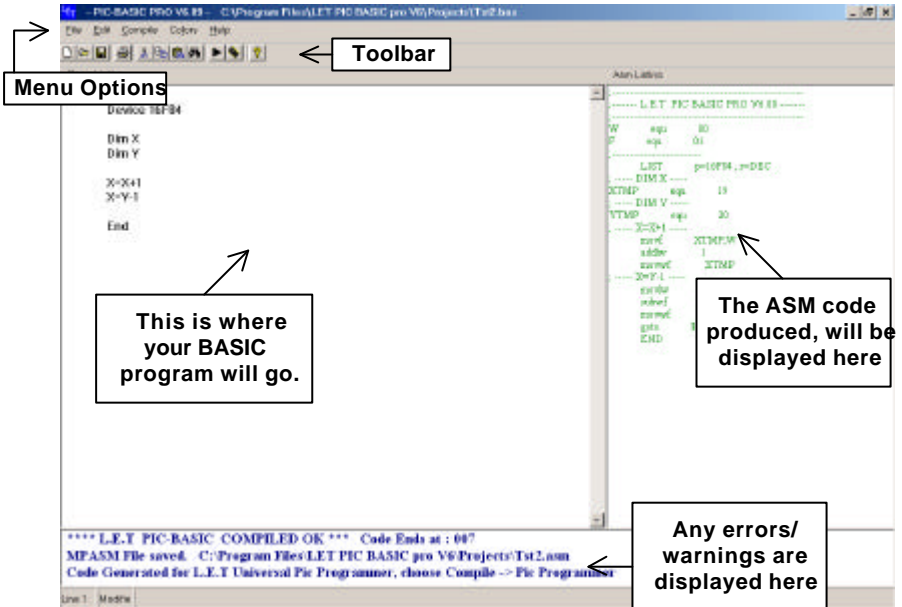
Note, the software is now fully installed on your hard drive so there is no need to put the CD in when you want to run it.

Important : This software was designed to be used at a resolution of 1024x768 or higher.

A resolution of 800x600 is still usable, however, you will probably find it restricting. Try to change to as high a resolution as possible. The IDEAL resolution is 1152x864.

2.2 - OK what now?

Throughout the rest of this chapter, we will be describing the Pic BASIC Pro. The Lite and Pro are essentially the same, front-end wise, with the Pro having a few extra options. If it is the Lite you are running, simply ignore sections related to the Pro version. Run the program by clicking **Start->Programs->Pic Basic Pro (or Lite)**. You will be faced with a screen that looks something like this.



You can choose to leap in and write a program if you wish (*not recommended*) or load a sample file (*recommended*). To load a file select **File->Open** or click the open folder icon in the toolbar. For now, load in the **test.bas** file. Note this program does not actually do anything but is useful to show you syntaxes, layout, and allows you to play with the editor. It will compile however.

To compile it, either select **Compile->Compile Basic** or click on the play button (*to the right of the binoculars*) on the toolbar. Very quickly, a stream of asm code is produced in the right hand window, and information about the compiled code appears in the bottom section. In your picbasic directory now is an asm file, which can be loaded into MPASM if you wish to use a third party PIC programmer.

However, if you have our Universal Pic Programmer, you can program the code directly into a PIC now by connecting our Pic Programmer, inserting the device, and then selecting **Compile->Pic Programmer** or clicking the Pic chip on the toolbar. That's the basics - you can now load/edit a file, compile it, and program it to a device if you wish

2.3 - What do all the other menu options do?

Most of the menu options are pretty much self-explanatory, but one needs mentioning a little further: -

The **File** menu allows you to load, save, and create basic files.

The **Edit** menu enables you to cut, copy, paste, and find/replace in your basic source file.

The **Colors** menu allows you to customise your window and font colors.

The **Help** menu allows you to have a quick-view of syntaxes while you're working and also to view version and contact info.

The most important menu is the Compile menu. Three options are provided for compile setup options. These are: -

- **Produce MPASM file** - If this option is selected an assn file will be produced in your Pic BASIC directory when you compile. This asm file can be loaded directly into MPASM file to produce PIC code suitable for a third-party PIC programmer.
- **Show op-codes** - Upon compiling, the op-codes are displayed in the right window along with the assembly code produced
- **ISP download** - Dump the contents directly into our In-Circuit serial programmer for testing.

The other two options in the **Compile** menu are **compile basic** which sets the compiler in motion, and **Pic Programmer**, which dumps the code directly to a PIC in our Pic Programmer.

Don't be afraid to play with the editor and menu options. You can't do any damage. The icons on the toolbar correspond to key features in the menu - if you are unsure as to what an icon does, simply leave the mouse pointer over it for a second and a quick description of the icon will appear in a yellow box.

3 - Program Rules

There are a few simple rules you must follow when producing a BASIC program using the LET Pic BASIC.

3.1 - Program Order

All programs must follow the structure below in this order: -

```
DEVICE { device }  
INCLUDE { packages }  
DIM { variables }  
SYMBOL { symbol }={ port.pin }  
DEFINE { port }={ input/output }  
INIT { packages,pins,ports }  
DATA { tables }  
{ ... ..  
... ..  
Your program  
... ..  
... .. }  
END
```

Obviously, not all these have to be included in every program. It will depend on your application, but when using any of the above, make sure they appear in this order. There is some slight flexibility e.g. **SYMBOL** can appear before **DIM** but as a rule, use the structure above.

3.2 - Labels

User defined labels **MUST** appear at the start of a line and be immediately followed by a colon (:) and space. For example:

```
A=3  
B=4  
Mylab: C=A+ B
```

3.3 -General Statements

Each line **MUST** begin with a TAB, unless it is a label (see above). Multiple statements on a line must have a colon (:) separating them and a space either side of the colon. For example: A=3 : B=4

3.4 - General hints on programming

- Avoid the use of too many **GOTO** statements, as this makes code unreadable and harder to debug.
- Use plenty of **REM** statements - you may know what you are doing now but will you still understand your code in 6 months time?
- Where possible use **GOSUB** routines to save PIC code space. e.g. The **DE-LAYMS** command uses between 8 and 12 bytes of PIC code memory, therefore, calling a delay 'subroutine' more than twice will save a lot of space.
- Try to give your labels meaningful names.
- Save your program before compiling.
- Save your program regularly, there is nothing worse than losing a day's work because the dog pulls the cable out or you have a power cut or (unusually!) windows crashes.
- Do not 'quick fix' bugs, always try to find out what's causing them as opposed to just 'patching'.
- Make use of the **SYMBOL** statement to give your pins meaningful names e.g. LED is a lot more understandable than B.4
- Use the ' :' character to split two lines of code into one. For example, the lines: -

```
A=MyVar  
GOSUB DoMyVar
```

Could be implemented on one line like this: -

```
A=MyVar : GOSUB DoMyVar
```

- **NOTE.** Command lines may not exceed 100 characters in length. If the maximum amount is exceeded, then unpredictable error messages will appear.

4 - Mathematical Operators and Comparators

4.1 - Mathematical Operators

LET Pic BASIC supports the following mathematical operators: -

Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide
&	Bitwise AND
	Bitwise OR
!	Bitwise XOR
<<	Shift Left
>>	Shift Right

All calculations are 8-bit integer types, any calculation that produces a result larger than 255 will not yield a correct result.

Add, Subtract, and Multiply need no explanation, we hope!

Divide will remove anything after the decimal point. e.g. 6/4 will return 1

AND, OR, and XOR perform a bit operation corresponding to the following logic tables: -

AND		
A	B	Result
0	0	0
0	1	0
1	0	0
1	1	1

OR		
A	B	Result
0	0	0
0	1	1
1	0	1
1	1	1

XOR		
A	B	Result
0	0	0
0	1	1
1	0	1
1	1	0

Shift Left and Shift Right, shift a value either way by a specific number of bits, to create a new value. For example:

- A=12 ' A contains the value 12, which is 00001100 in binary
- A=A<<4 ' A is shifted four times to the left, which leaves 11000000 in binary. Variable A now contains the value 192.
- A=A>>3 ' A is shifted three times to the right, which now leaves ' 00011000 in binary.
- ' Variable A now contains the value 24

4.1 - Comparators

When using the **IF** statement, the following comparators are available: -

Comparator	Description
=	Is equal to
<	Is less than
>	Is greater than
<>	Is not equal to
<=	Is less than OR equal to
>=	Is greater than OR equal to
=<	Is less than OR equal to
=>	Is greater than OR equal to

It can be seen from the above table that <= and =< are the same comparator. This is also true for >= and =>.

Examples :

Assuming A=3, B=5, C=5, and D=10

- IF A < B THEN { code }** ' A is less than B so: TRUE, Code performed
- IF A > D THEN { code }** ' A is not greater than D so: FALSE, Code not performed
- IF B = C THEN { code }** ' B is equal to C so: TRUE, Code performed
- IF C >= B THEN { code }** ' C is equal to B so: TRUE, Code performed
- IF A <= B THEN { code }** ' D is not less than or equal A: FALSE, Code not performed

5 - L.E.T Pic BASIC commandsCommands in **bold** apply to the LET Pic BASIC **Pro** only.

- 5.1 **ADIN**
- 5.2 ASM
- 5.3 **BSTART**
- 5.4 **BSTOP**
- 5.5 **BUSIN**
- 5.6 **BUSOUT**
- 5.7 BUTTON
- 5.8 CLEAR (or LOW)
- 5.9 **CLS**
- 5.10 **COUNTER**
- 5.11 **CURSOR**
- 5.12 DATA
- 5.13 DEFINE
- 5.14 DELAYMS
- 5.15 DELAYUS
- 5.16 DEVICE
- 5.17 DIM
- 5.18 **EEDATA**
- 5.19 END
- 5.20 FOR....TO....NEXT....[STEP]
- 5.21 GOSUB....RETURN
- 5.22 GOTO
- 5.23 IF....THEN
- 5.24 **INCLUDE**
- 5.25 **INIT**
- 5.26 **INKEY**
- 5.27 INPORTA , INPORTB , INPORTC
- 5.28 [LET]
- 5.29 **MEMREAD**
- 5.30 **MEMWRITE**
- 5.31 OUTA , OUTB , OUTC
- 5.32 PEEK
- 5.33 POKE
- 5.34 **PRINT**
- 5.35 READ
- 5.36 REM
- 5.37 RESTORE
- 5.38 **RSIN**
- 5.39 **RSOUT**
- 5.40 SET (or HIGH)
- 5.41 SLEEP
- 5.42 SOUND
- 5.43 STOP
- 5.44 **STORE**
- 5.45 SWAP
- 5.46 SYMBOL
- 5.47 **TIMER**

5.1 ADIN

Syntax : { *variable* } = **ADIN**({ *channel number* })

Overview : Read the value of the Analogue to Digital Converter on the 16C71.

Operators : *variable* is a user defined variable.
channel number must be a numeric value between 0 and 3

Example : ‘ Retrieve the value of channel 3
 ‘ of the A to D Converter and place in variable A.

```
INCLUDE A2D
INIT A2D
DIM A
DEFINE PortA=00001000 ‘ Configure AN3 (PortA.3) as an input
A=ADIN (3)           ‘ Place the conversion into variable A
```

Notes : This command only applies to the PIC16C71. Thus, it is NOT available in the PLUS version of the compiler.

Although the PIC powers up with the port pins set as inputs, it is always wise to manually configure the ports with the **DEFINE** command

If multiple conversions are being implemented, the a small delay should be used after the **ADIN** command. This allows the ADC’s internal capacitors to discharge fully: -

```
Again: A=ADIN (3)      ‘ Place the conversion into variable A
PAUSEUS (1)          ‘ pause for 4us
GOTO Again           ‘ Read the ADC forever
```

Package : The A2D package must first be loaded and initialised before this command is available.

See also : **INCLUDE, INIT**

5.2 ASM

Syntax : **ASM** {
 assembler mnemonics
 }

or

ASM { *assembler mnemonic* }

Overview : Incorporate inline assembler in the BASIC code.

Operators : *assembler mnemonics* refer to assembler commands for the selected device. This goes outside the scope of the manual so check Microchip data sheets for more information.

Notes : Requires MPASM (*from Microchip technologies*) to assemble the inline mnemonics for the LITE version. The PRO has a built-in assembler therefore MPASM is not required.

If a single assembler instruction is required i.e. NOP. Then the mnemonic may be surrounded by the curly brackets: -

ASM { NOP }

ASM { CLRWDT }

A space should be left between the curly brackets and the mnemonic.

5.3 BSTART

Syntax : **BSTART**

Overview : Part of the standard interface to the I²C bus, which sends a START condition across the I²C bus.

Notes : The I²C commands use fixed pins for the SDA and SCL connections:-
PortA bit 0 is used for SDA
PortA bit 1 is used for SCL

When the I2CBUS commands are used, PortA bits 0 and 1 are automatically configured as outputs.

If serial commands are used on the same port (i.e. PortA) then the **INIT I2CBUS** command must be placed after the **INIT SERIAL** command.

Package : The I2CBUS package must first be loaded before this command is available:-

INCLUDE I2CBUS

See also : **BSTOP, BUSIN, see BUSOUT for suitable circuit, INIT, INCLUDE**

5.4 BSTOP

Syntax : **BSTOP**

Overview : Part of the standard interface to the I²C bus, which sends a STOP condition across the I²C bus.

Notes : The I²C commands use fixed pins for the SDA and SCL connections:-
PortA bit 0 is used for SDA
PortA bit 1 is used for SCL

When the I2CBUS commands are used, PortA bits 0 and 1 are automatically configured as outputs, regardless of the value used in the **DEFINE** command.

If serial commands are used on the same port (i.e. PortA) then the **INIT I2CBUS** command must be placed after the **INIT SERIAL** command.

Package : The I2CBUS package must first be loaded before this command is available:-

INCLUDE I2CBUS

See also : **BSTART, BUSIN, see BUSOUT for suitable circuit, INIT, INCLUDE**

5.5 BUSIN

Syntax : { *variable* } = **BUSIN**

Overview : Receives a byte from the I²C bus.

Operators : *variable* is a user defined variable.

Example : ‘ Receive a byte from the I²C bus
 ‘ and placed it into the variable A.

```
INCLUDE I2CBUS  
DIM A  
INIT I2CBUS  
A=BUSIN
```

Notes : The I²C commands use fixed pins for the SDA and SCL connections:-
 PortA bit 0 is used for SDA
 PortA bit 1 is used for SCL

When the I2CBUS commands are used, PortA bits 0 and 1 are automatically configured as outputs, regardless of the value used in the **DEFINE** command.

If serial commands are used on the same port (i.e. PortA) then the **INIT** I2CBUS command must be placed after the **INIT** SERIAL command.

Package : The I2CBUS package must first be loaded before this command is available:-

```
INCLUDE I2CBUS
```

See also : **BSTART, BTOP, see BUSOUT for suitable circuit, INIT, INCLUDE**

5.6 BUSOUT

Syntax : **BUSOUT** (*{ number / variable / expression }*)

Overview : Output a *number, variable, or expression* to the I²C bus.

Example : ' Send the contents of variable A+36 across the I²C bus.

```

INCLUDE I2CBUS
DIM A
INIT I2CBUS
A=25
BUSOUT (36+A)
    
```

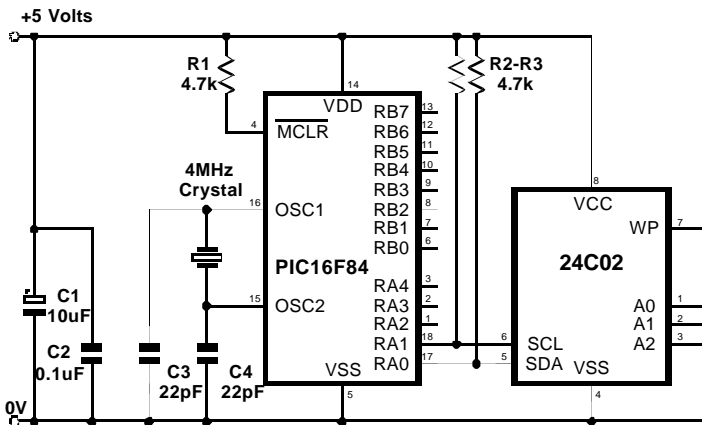
Notes : The I²C commands use fixed pins for the SDA and SCL connections:-
 PortA bit 0 is used for SDA
 PortA bit 1 is used for SCL

When the I2CBUS commands are used, PortA bits 0 and 1 are automatically configured as outputs.

If serial commands are used on the same port (i.e. PortA) then the **INIT** I2CBUS command must be placed after the **INIT** SERIAL command.

Package : The I2CBUS package must first be loaded before this command is available:-

See also : **BSTOP, BSTART, BUSIN**



A typical use for the I²C commands is for interfacing with serial eeproms. The above diagram shows the connections to the I²C bus of a 24C02 serial eeprom.

5.7 BUTTON

Syntax : **BUTTON** { *port.bit / symbol* }

Overview : Program execution is halted while the software waits for the pin used in **BUTTON** to invert its current state i.e. high to low or low to high

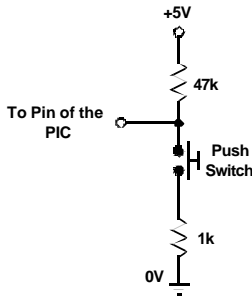
Operators : *port* can be A, B, or C
bit is the pin to await inversion
symbol is a symbolic representation of the pin

Example :

```

DEFINE PORTB = 11111101      ' Set PORTB's direction
SYMBOL LED=B.1              ' Assign the LED to PortB Bit-1
SYMBOL KEY=B.0              ' Assign the push switch to PortB Bit-0
Loop: HIGH LED                ' Illuminate the LED
      BUTTON KEY              ' Wait for a keypress
      LOW LED                 ' Then extinguish the LED
      BUTTON B.0              ' Wait for another keypress
      GOTO LOOP              ' And do it all over again
      END                    ; The mandatory END statement
    
```

Notes : The pin to be used for **BUTTON** must have been previously defined as an input



The above diagram shows one possible connection of a push switch to the PIC. A logic low will be produced by activating the switch. The 47k resistor stops the pin from floating while the switch is open. This may be eliminated if the internal PortB pullup resistors are enabled. The 1k resistors eliminates any shorts occurring if the switch is inadvertently closed while the pin is configured as an output.

See also : **DEFINE, SYMBOL**

5.8 CLEAR (or LOW)

Syntax : CLEAR (or LOW) { port.bit / symbol }

Overview : Place a port pin in the logic low state - i.e. 0

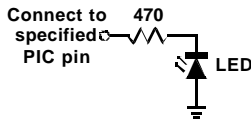
Operators : port can be A, B, or C
 bit is the bit number to be cleared to 0
 symbol can be a symbolic representation of a port pin

Example : ‘ Bit 2 on port B is **CLEAR**ed to 0. Bit 4 on port B (symbolised by the ‘ name LED) is **SET** to 1. Bit 3 on port B is **CLEAR**ed to 0.

SYMBOL LED=B.4	‘ Assign the LED to PortB Bit-4
CLEAR B.2	‘ Pull LOW PortB Bit-2
SET LED	‘ Set HIGH PortB Bit-4
LOW B.3	‘ Pull LOW PortB Bit-3

Notes : There is no difference between the **CLEAR** and **LOW** commands. The pin to be used for **CLEAR** or **LOW** must have been previously defined as an **output**

See also : SET, DEFINE, SYMBOL



The above diagram shows the connection of an LED to any of the pins of a PIC. A resistor must be used in series with the LED to limit the current supplied to it.

5.9 CLS

Syntax : CLS

Overview : Part of the LCD package that clears the LCD and places the cursor at the home position i.e. column 1, row 1

Example :

```

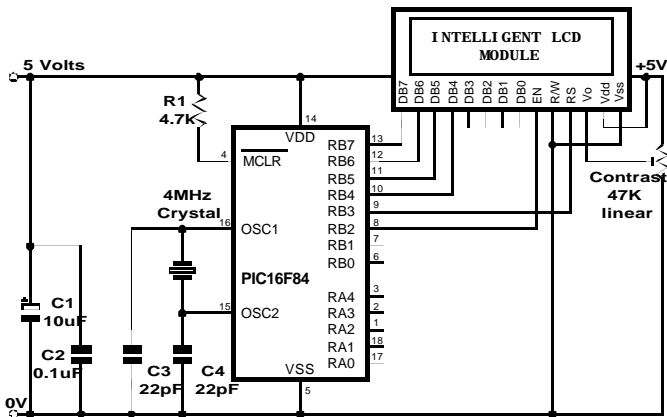
DEVICE 16F84           ; Use the PIC16F84 micro
INCLUDE LCD           ; Load the LCD routines
INIT LCD,PortB       ; Place the LCD on PortB
CLS                  ; Clear the LCD
PRINT "HELLO"        ; Display the word "HELLO" on the LCD
CURSOR 1,2          ; Move the cursor to line 2, position 5
PRINT "WORLD"       ; Display the word "WORLD" on the LCD
END                  ; The mandatory END statement
    
```

Firstly the LCD is cleared using the **CLS** command, which also places the cursor at the home position i.e. column 1, row 1. Next, the word HELLO is displayed in the top left corner. The cursor is then moved to column 1 row 2, and the word WORLD is displayed.

Notes : The connection of the LCD uses fixed pins. If PortB is chosen then the LCD's data lines connect to PortB bits 4..7. The LCD's RS line connects to PortB bit-3, and the LCD's EN line connects to PortB bit-2. If PortC is chosen then the same bits of the port apply. The diagram below illustrates a typical connection of an LCD to PortB.

Package : The LCD package must first be loaded then initialised before this command is available. See above example.

See also : CURSOR, PRINT, INIT, INCLUDE



5.10 COUNTER

Syntax : **COUNTER** { *on/off* },{ *high/low* }
 { *variable* }=**COUNTER**

Overview : Clears and then enables a counter in from the RTCC pin on a PIC
 Use { *variable* }=**COUNTER** to read.

Operators : *on* – clear and enable the counter
 off – stop the counter
 high – counter will increment on high to low transition
 low – counter will increment on low to high transition
 variable is a user-defined variable

Example : ‘ Flashes an LED on and off
 ‘ Use RA4 as counter input pin
 ‘ LED on / off every 128 counts
 ‘ LED on PortB.0

```

DEVICE 16F84           ; Use the PIC16F84 micro DIM A,B
DEFINE PortB=11111110 ; Configure PortB's direction
DEFINE PortA=11111111 ; Configure PortA's direction
SYMBOL LED=B.0       ; Assign the LED to PortB Bit 0
COUNTER On,High
Loop: A=COUNTER       ; Variable A now contains the Count
IF A>128 then SET LED ; If A> than 128 then illuminate the LED
IF A<128 then CLEAR LED ; If A> than 128 then extinguish the LED
GOTO Loop             ; Do it indefinitely
END                  ; The mandatory END statement
  
```

See also : **DEFINE**

5.11 CURSOR

Syntax : **CURSOR** { *command / column,row* }

Overview : Move the cursor position on the LCD to a location corresponding to *column,row* or moves it using a *command*

Operators : *command* can be one of the following:

- LEFT - move the cursor left a character
- RIGHT - move the cursor right a position
- HOME - move the cursor to the top left i.e. column 1, row 1

column is the column number from 1 to maximum columns
row is the row number from 1 to maximum rows

Example :

DEVICE 16F84	;	Use the PIC16F84 micro
INCLUDE LCD	;	Load the LCD routines
INIT LCD,PortB	;	Place the LCD on PortB
CLS	;	Clear the LCD
PRINT "HELLO"	;	Display the word "HELLO" on the LCD
CURSOR 1,2	;	Move the cursor to line 2, position 1
PRINT "WORLD"	;	Display the word "WORLD" on the LCD
END	;	The mandatory END statement

Firstly the LCD is cleared using the CLS command, which also places the cursor at the home position i.e. line 1, position 1. Next the word HELLO is displayed in the top left corner. The cursor is then moved to line 2 position 1 and the word WORLD is displayed.

Package : The LCD module must first be loaded then initialised before this command is available. See above example.

See also : **INIT, INCLUDE , PRINT, see CLS for LCD connection circuit**

5.12 DATA

Syntax : **DATA** { *alphanumeric data* }

Overview : **DATA** defines a table of alphanumeric data.

Operators : *alphanumeric data* can be any alphabetic character or string enclosed in quotes (") or numeric data without quotes.

Example : **DIM I**
DATA 5,8,"fred",12
RESTORE
READ I ' Variable I will now contain the value 5
READ I ' Variable I will now contain the value 8
' Pointer now placed at location 4 in our data table i.e. "r"
RESTORE 3
' I will now contain the value 114 i.e. the 'r' character in decimal
READ I

The data table is defined with the values 5,8,102,114,101,100,12 as "fred" equates to f:102, r:114, e:101, d:100 in decimal. The table pointer is immediately restored to the beginning of the table. This is not always required but as a general rule it is a good idea to prevent table reading from overflowing.

The first **READ I**, takes the first item of data from the table and increments the table pointer. The next **READ I** therefore takes the second item of data. **RESTORE 3** moves the table pointer to the fourth location (first location is pointer position 0) in the table - in this case where the letter 'r' is. **READ I** now retrieves the decimal equivalent of 'r' which is 114.

Notes : Alphanumeric data is allowed in LET Pic BASIC Pro. Numeric only in the LITE version.

DATA tables must be declared AFTER any **DIM** statements, but BEFORE any **INIT** commands. Attempts to read past the end of the table will result in errors and unpredictable results.

Only one **DATA** statement is allowed per program. If the alphanumeric contents of the **DATA** statement will not fit on one line then the extra information must be placed directly below the **DATA** statement after a trailing comma: -

```
DATA "HELLO",
      "WORLD"
```

is the same as: -

```
DATA "HELLO WORLD"
```

See also: **READ , RESTORE**

5.13 DEFINE

Syntax : **DEFINE** { *port* } = { *input* / *output* }

Overview : **DEFINE** a port using 8 bits to represent inputs or outputs for each bit

Operators : *port* can be PortA, PortB, or PortC
input is represented by a 1
output is represented by a 0

Example : ‘ PortB is defined as bits-7,6,5,4 as inputs and bits-3,2,1,0 as outputs
DEFINE PortB=11110000

Notes : Even though PortA does not use all 8 bits you **MUST** still use an 8-bit definition.

The first number in the **DEFINE** refers to bit-7 of the specified port.
DEFINE must be placed after any **DIM** statements and before any **DATA** tables.

5.14 DELAYMS

Syntax : **DELAYMS** ({ *length* })

Overview : Delay execution for *length* x milliseconds

Operators : *length* is a variable or number

Example : **DIM** B
 B=50
 DELAYMS (100)
 DELAYMS (B)

Notes : The delay commands assume a crystal of 10MHz. Therefore you will need to adjust the length for other crystal frequencies. For example: -

DELAYMS (100) will delay 100ms on a 10MHz oscillator
DELAYMS (40) will delay 100ms on a 4MHz oscillator

The compiler uses inline code for the **DELAYMS** command, therefore, each time the command is used, 9 bytes of memory are used up. This can accumulate to quite a bit of memory if more than one **DELAYMS** command is used. The best solution is to place the **DELAYMS** command within a subroutine: -

Pause: **DELAYMS** (DELAY)
 RETURN

The amount of time to delay (*in ms*) is placed into the variable DELAY, then a call is made to the subroutine PAUSE.

If the 16C5X range of PICs are used, care should be taken to ensure that no more than 2 subroutines are being used at the same time, as these devices only have a 2 level deep STACK.

See also : **DELAYUS**

5.15 DELAYUS

Syntax : **DELAYUS** ({ *length* })

Overview : Delay execution for *length* x microseconds

Operators : *length* is a variable or number

Example : **DIM B**
 B=50
 DELAYUS (100)
 DELAYUS (B)

Notes : The delay commands assume a crystal of 10MHz. Therefore you will need to adjust the length for other crystal frequencies. For example: -

DELAYUS (100) will delay 100us on a 10MHz oscillator

DELAYUS (40) will delay 100us on a 4MHz oscillator

The compiler uses inline code for the **DELAYUS** command, however, fewer bytes of memory are used for each **DELAYUS** command, therefore, very little saving would be accomplished by placing the **DELAYUS** command into a subroutine, as in the case of **DELAYMS**.

The minimum resolution using a 4MHz crystal is 4us. If smaller delays are required, then inline assembler is used: -

ASM { Clrwdt } ' Delay for 1us using a 4MHz crystal

ASM { ' Delay 2us using a 4MHz crystal
 Clrwdt
 Clrwdt
 }

Using the CLRWDT (*Clear Watchdog Timer*) instruction is recommended over the NOP instruction, as the former will allow the watchdog timer to be enabled at programming time.

See also : **DELAYMS**

5.16 DEVICE

Syntax : **DEVICE** { *device number* }

Overview : **DEVICE** is used to inform the compiler which device code must be produced.

Operators : *device number* takes one of the following values :

- 16C54
- 16C55
- 16C56
- 16C57
- 16C71
- 16C84
- 16F83
- 16F84
- 12C508
- 12C509

Notes : **DEVICE** must be the first command placed in the program.

5.17 DIM

Syntax : **DIM** { *variable* }

Overview : All user-defined variables must be declared using the **DIM** statement.

Operators : *variable* can be any alphabetic character or string.

Example 1 : **DIM** A,B,MyVar,fred,cat,zz

Example 2 :
DIM A
DIM B
DIM MyVar

Notes : **DIM** must be placed near the beginning of the program. Any references to variables not declared or before they are declared will produce errors.

Variables are all 8-bits in length, which means that any single variable may contain the value 0 to 255.

Do not use variable names more than 12 characters long.

Variable names should be purely alphabetic. Alphanumeric variable names are not allowed. i.e.

DIM Fred is **VALID**

DIM Fred2 is **INVALID**

Variable names are case insensitive, which means that the variable: -

DIM MyVaR

Is the same as...

DIM MYVAR

5.18 EEDATA

Syntax : { *variable* }=**EEDATA** ({ *address* })

Overview : Read data from the internal eeprom of a 16C84 or 16F84

Operators : *variable* is a user-defined variable
address can be a user-defined variable or numeric value, and is the location in the eeprom from 0-63

Example : **INCLUDE** EEPROM
 INIT EEPROM
 DIM A,B,C
 C=4
 A=**EEDATA** (10)
 B=**EEDATA** (C+8)

Variable A contains the value of data at position 10 in the eeprom
Variable B contains the value of data at position 12 in the eeprom

Notes : This command only applies to the 16C84 or 16F84

Package : The EEPROM module must first be loaded and initialised before the **EEDATA** command is available: -

INCLUDE EEPROM
INIT EEPROM

See also : **STORE, INIT, INCLUDE**

5.19 END

Syntax : **END**

Overview : The **END** statement stops compilation of source. Nothing in the BASIC source after an **END** is compiled.

Notes : **END** stops the PIC processing by putting it into a continuous loop. The port pins remain the same but the device is **NOT** in low power mode.

See also : **STOP, SLEEP**

5.20 FOR....TO....NEXT

Syntax : **FOR** {*variable*} = {*count*} **TO** {*endcount*}
 {*code body*}
 NEXT {*variable*}

Overview : The **FOR....NEXT** loop is used to execute a statement or series of statements many times.

Operators : *variable* refers to an index variable used for the sake of the loop. This index variable can itself be used in the code body but beware of altering its value within the loop as this can cause many problems.
count is the start number of the loop, which will initially be assigned to the *variable*. This does not have to be an actual number - it could be the contents of another variable.
endcount is the number on which the loop will finish. Note that this number will be used in the loop before exiting. This does not have to be an actual number - it could be the contents of another variable.

Example : A=0 : B=1
 FOR X=B TO 5
 FOR Y=1 TO 10
 A=A+1
 NEXT Y
 NEXT X

This shows how loops can be embedded. The Y loop will be performed 5 times because it is contained within the X loop which goes from the contents of B (i.e. 1) to 5. Thus, after the first pass, A will contain 10. The second pass will change A to 20, the third to 30, the fourth to 40 and the loops will finish with A containing 50.

Notes : Each **NEXT** *variable* must be the same *variable* as the **FOR** *variable* that precedes it when embedding loops.

5.21 GOSUB....RETURN

Syntax : **GOSUB** { *label* }
 ...
 ...
 RETURN

Overview : **GOSUB** jumps the program to a defined label and continues execution from there. Once the program hits a **RETURN** command the program returns to the **GOSUB** that called it and continues execution from that point.

Operators : *label* is a user-defined label placed at the beginning of a line which must have a colon ':' directly after it.

Example : **GOSUB** SubA
 GOSUB SubB
 STOP

```
SubA: { subroutine A code  
.....  
.....  
}  
RETURN
```

```
SubB: { subroutine B code  
.....  
.....  
}  
RETURN  
END
```

This example gives a good idea of structuring your programs. By placing the subroutines at the end of the program with a **STOP** before them, you are ensuring that they can only ever be called by a **GOSUB** command as program execution will never reach them naturally. The first subroutine is called and executed. The **RETURN** command sends execution back and then the second subroutine is called. This is extremely useful for routines that need to be called many times from different parts of the program.

Notes : Make sure labels are placed at the beginning of a line with no spaces in front and have a colon ':' directly after.
 Labels must only contain alphabetic characters.

5.22 GOTO

Syntax : **GOTO** { *label* }

Overview : **GOTO** jumps the program to a defined label and continues execution from there.

Operators : *label* is a user-defined label placed at the beginning of a line which must have a colon ':' directly after it.

Example : **IF** A=3 **THEN GOTO** Jumpover
{ *code here executed only if A<>3*
.....
.....
}
Jumpover: {*continue code execution*}

In this example, if A=3 then the program jumps over all the code below it until it reaches the *label* jumpover where program execution continues as normal.

Notes : Make sure labels are placed at the beginning of a line with no spaces in front and have a colon ':' directly after them.
Labels must only contain alphabetic characters: -

LABEL: is **VALID**

LABEL1: is **INVALID**

5.23 IF....THEN

Syntax : **IF** { *comparison* } **THEN** { *expression* }

Overview : Evaluates the *comparison* and, if it fulfils the criteria, executes *expression*. If *comparison* is not fulfilled the *expression* is ignored

Operators : *comparison* is composed of variables, numbers and comparators as mentioned in **section 4.2**
expression is the statement to be executed should the *comparison* fulfil the **IF** criteria

Example 1 : **SYMBOL** LED=B.4
 A=3
 SET LED
 IF A>4 **THEN** **CLEAR** LED

In the above example, A is not greater than 4 so the **IF** criteria isn't fulfilled. Consequently, the **CLEAR** LED statement is never executed leaving the state of port pin B.4 high.

Example 2 : A=4 : B=4 : C=10
 IF A>=B **THEN** C=C*2

In example 2, variable A is **not** greater than B but it **is** equal to it, thus the **IF** criteria is fulfilled. Consequently, variable C is multiplied by 2.

Example 3 : **IF** INPORTA & 1=1 **THEN** X=X | 2

Example 3 illustrates a method for testing individual bits of a port. If bit-1 of PortA is equal to 1 (logic high) then bit-1 of variable X is set.

5.24 INCLUDE

Syntax : **INCLUDE** { *package* }

Overview : Use the **INCLUDE** statement to assign variables and information from our predefined packages.

Operators : *package* can be any one or more of the following : -

- LCD
- KEYPAD
- A2D
- I2CBUS
- EEPROM
- SERIAL

Example : **INCLUDE** LCD,KEYPAD
Includes the routines required for LCD and KEYPAD operation.

Notes : **INCLUDE** must be placed after the **DEVICE** statement. If you include packages but fail to use them, you are wasting valuable memory.

See also : **INIT**

5.25 INIT

Syntax : `INIT { package },{ extra info }`

Overview : Use the **INIT** to assign code from our predefined packages to your program.

Operators : *extra info* depends on which package you are using for **INIT** - this is summarised below :

Package	Extra info Required
LCD	PortB or PortC
KEYPAD	PortB or PortC
A2D	None
I2CBUS	None
EEPROM	None
SERIAL	RSin pin, RSout pin, [Dtr] optional

Example :

```

INCLUDE LCD,KEYPAD,A2D,I2CBUS,EEPROM,SERIAL
SYMBOL Rin=A.0
SYMBOL Rout=A.1
SYMBOL Dtr=A.2
INIT LCD,PortB
INIT KEYPAD,PortC
INIT SERIAL Rin,Rout,Dtr
INIT I2CBUS
INIT A2D
    
```

Notes : When using **INIT** with the serial package you do NOT use a comma after the word serial - you use a **SPACE** instead.

A **DEFINE** command is required before the A2D, I2CBUS, and SERIAL **INIT** commands are used. The **DEFINE** command is not required for the LCD, EEPROM, or KEYPAD **INIT** commands

See also : **INCLUDE**

5.26 INKEY

Syntax : { *variable* } = INKEY

Overview : Wait for key press on keypad and place value in *variable*

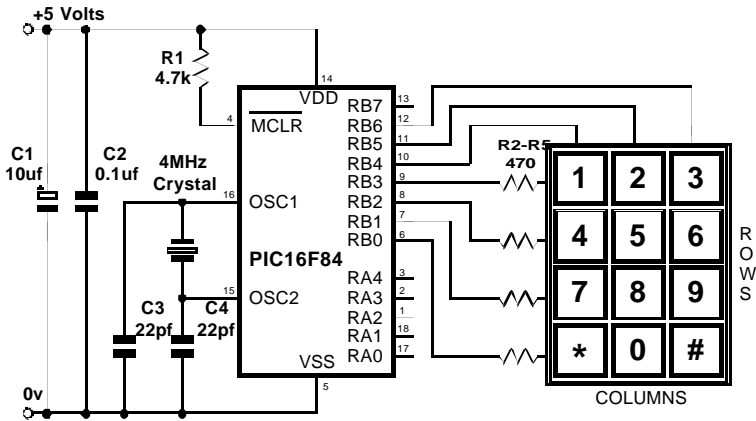
Operators : *variable* is a user defined variable

Example : **INCLUDE** KEYPAD
INIT KEYPAD,PortB ' Assign the keypad to PortB
DIM A
A=INKEY

Notes : The keypad connection uses fixed pins for the Row and Column lines. These must be connected to PortB or PortC (*if available*).

Package : The KEYPAD package must first be loaded and initialised, before the **INKEY** command is available. See the above example.

See also : **INIT, INCLUDE**



The above diagram illustrates a typical connection of a 12-button keypad to a PIC16F84. If a 16-button type is used, then COLUMN 4 will connect to PortB.7 (RB7).

5.27 INPORTA , INPORTB , INPORTC

Syntax : { *variable* } = **INPORTA** or **INPORTB** or **INPORTC**

Overview : Get data from PortA, PortB, or PortC and place into *variable*

Operators : *variable* is a user defined variable

Example 1 : ‘ Place the 8-bit contents of PortA into the variable X
DEFINE PortA=11111111
X=**INPORTA**

Example 2 : ‘ Place the contents of PortB, bitwise anded with 4, into variable Y
‘ This has the result of masking all but bit-3 of the port
DEFINE PortB=11001111
Y=**INPORTB** & 4

Notes : Before the **INPORT** command is used, the port’s direction should be configured using the **DEFINE** command.

See also : **OUTA, OUTB, OUTC, DEFINE**

5.28 [LET]

Syntax : [LET] { *variable* } = { *expression* }

Overview : Assigns an *expression* to a *variable*

Operators : *variable* is a user defined variable.
expression is one of many options – these can be a combination of variables, maths, and numbers or other command calls (see below)

Example 1 : LET A=1
A=1
Both the above statements are the same

Example 2 : A=B+3

Example 3 : A=A<<1

Example 4 : LET B=EEDATA (C+8)

Notes : The LET command is optional

See also : DIM

5.29 MEMREAD

Syntax : { *variable* } = **MEMREAD** { *location* }

Overview : Reads data from an external 24C01 / 02 / 04 / 08 serial eeprom connected to the I²C bus

Operators : *variable* is a user defined variable
location is a user defined variable or constant which points to the location in the eeprom you wish to read

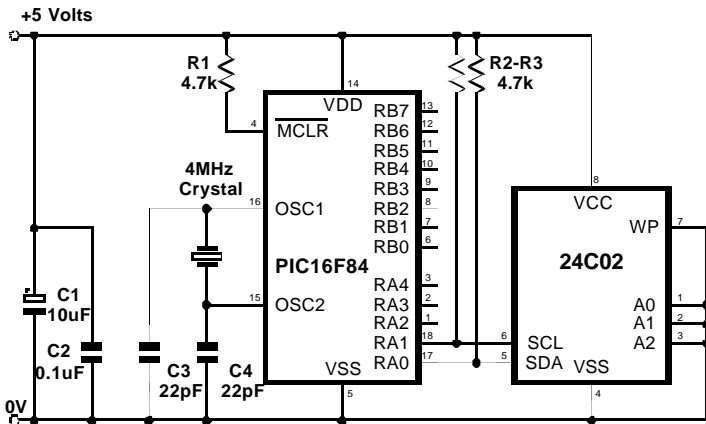
Example : **INCLUDE I2CBUS**
DIM A
INIT I2CBUS
A=MEMREAD (120)

This reads location 120 of the serial eeprom attached, and places the value in variable A

Package : The **MEMREAD** command is only available after the I2CBUS package is loaded and initialised

Notes : The I²C commands use fixed pins for the SDA and SCL connections:-
PortA bit 0 is used for SDA
PortA bit 1 is used for SCL

See also : **MEMWRITE, INIT**



The above circuit shows the connections for a 24C02 serial eeprom. Eeproms, 24C01, 24C04, and 24C08 also use this circuit.

5.30 MEMWRITE

Syntax : **MEMWRITE** { *location* }, { *value* }

Overview : Write data to an external 24C01 / 02 / 04 / 08 serial eeprom connected to the I²C bus

Operators : *value* is a variable or a number
location is a user defined variable or constant which points to the location in the eeprom you wish to write

Example :

```

INCLUDE I2CBUS
DIM A
DIM Addr
INIT I2CBUS
A=10
Addr=121
MEMWRITE 120,5           ' Write 5 into address location 120
DELAY (4)                ' Delay for approx 10ms
MEMWRITE Addr,A         ' Write the value of A into address 121
DELAY (4)                ' Delay for approx 10ms
    
```

This writes location 120 of the serial eeprom attached with the value 5 and location 121 with the contents of A, in this case 10

Package : The **MEMWRITE** command is only available after the I2CBUS package is loaded and initialised

Notes : The I²C commands use fixed pins for the SDA and SCL connections:-
 PortA bit 0 is used for SDA
 PortA bit 1 is used for SCL

After the **MEMWRITE** command is used, a delay of 10ms must be implemented to allow the serial eeprom to allocate the byte into memory.

If serial commands are used on the same port (i.e. PortA) then the **INIT** I2CBUS command must be placed after the **INIT** SERIAL command.

See also : **MEMWRITE, INIT, see MEMREAD for a suitable circuit layout**

5.31 OUTA , OUTB , OUTC

Syntax : **OUTA** or **OUTB** or **OUTC** ({ *data* })

Overview : Send data to PortA, PortB, or PortC

Operators : *data* is a user defined variable or number

Example : **DIM X** ‘ Declare variable X
 X=32 ‘ Place 32 into variable X
 OUTA (12) ‘ Place the value 12 (00001100) onto PortA
 OUTB (X) ‘ Place the contents of variable X onto PortB

Notes : Before the **OUT** command is used, the port’s direction should be configured using the **DEFINE** command.

See also : **INPORTA, INPORTB, INPORTC, DEFINE**

5.32 PEEK

Syntax : { *variable* } = **PEEK** ({ *file register* })

Overview : Use the **PEEK** command to retrieve the value of a File Register and place into a variable

Operators : *variable* is a user defined variable.
 file register can be a number or the contents of a variable.

Example 1 : A=PEEK (15)

Variable A will contain the value of File Register 15. If the device is a 16F84, for example, this file register is one of the 36 general-purpose registers (SRAM).

Example 2 : B=15
 A=PEEK (B)

Same function as example 1

See also : **POKE**

5.33 POKE

Syntax : **POKE** ({ *data* },{ *file register* })

Overview : Use the **POKE** command to assign data to a File Register.

Operators : *data* can be a number or the contents of a variable.
file register can be a number or the contents of a variable.

Example : ‘ File Register 12 will be assigned the value 15.
A=15
POKE (12,A)

Notes : **POKE** and **PEEK** are the two most powerful commands in the compiler’s arsenal. With proper use, these commands allow full control over the PIC’s hardware registers.

See also : **PEEK**

5.34 PRINT

Syntax : **PRINT** { *string* / [**\$**][**#**] *variable* / [**\$**][**#**] *number* }

Overview : Display *strings, variables or numbers* on an LCD display

Operators : *string* is any alphanumeric string you wish to display which must be enclosed in quotes ("").
variable is a user defined variable.
number is (guess what) a number.

The # and \$ symbols define how the output is displayed.
means display the *number/variable* as a character
\$ means display the *number/variable* as a hex number.
 if our *variable* contains the value 70 then :

PRINT *variable* will display 70 (i.e. the value)
PRINT **#***variable* will display a letter 'F' (i.e. ASCII char 70)
PRINT **\$***variable* will display 46 (i.e. 70 in hex)

Example : **DEVICE** 16F84
INCLUDE LCD
DIM MyVar
INIT LCD,PortB
CLS
 MyVar=65
PRINT "HELLO"
CURSOR 5,2
PRINT \$MyVar,#32,#Myvar
STOP
END

Firstly, the LCD is cleared using the **CLS** command, placing the cursor at the home position i.e. 1,1. Next, the word HELLO is displayed in the top left corner. The cursor is then moved to column 5 row 2. Output then is 41 (65 converted to hex), a space (ASCII char 32), letter 'A' (ASCII char 65)

Package : The LCD package must first be loaded then initialised before this command is available. See above example.

See also : **INIT, INCLUDE, CURSOR, see CLS for a suitable circuit**

5.35 READ

Syntax : **READ** { *variable* }

Overview : **READ** the next value from a **DATA** table and place into *variable*

Operators : *variable* is a user defined variable

Example : **DIM I**
DATA 5,8,"fred",12
RESTORE
READ I
' I will now contain the value 5
READ I
' I will now contain the value 8
RESTORE 3
' Pointer now placed at location 4 in our data table i.e. "r"
READ I
' I will now contain the value 114 i.e. the 'r' character in decimal

The data table is defined with the values 5,8,102,114,101,100,12 as "fred" equates to f:102,r:114,e:101,d:100 in decimal. The table pointer is immediately restored to the beginning of the table. This is not always required but as a general rule, it is a good idea to prevent table reading from overflowing.

The first **READ I** takes the first item of data from the table and increments the table pointer. The next **READ I** therefore takes the second item of data.

RESTORE 3 moves the table pointer to the fourth location in the table – in this case where the letter 'r' is. **READ I** now retrieves the decimal equivalent of 'r' which is 114.

Notes : Alphabetic **DATA** is only allowed in LET Pic BASIC Pro. Numeric only in the LITE version.

DATA tables must be declared **after** any **DIM** statements. Attempts to read past the end of the table will result in errors and undetermined results.

See also : **DATA , RESTORE**

5.36 REM

Syntax : **REM** *comments* or ' *comments*

Overview : Insert reminders in your BASIC source code. These lines are not compiled and are used merely to provide information to the person viewing the source.

Operators : *comments* can be anything

Example : **DIM** A,B,C
 A=12 : B=4
 REM Now I'm going to add them together
 C=A+B
 ' Now I'm going to subtract them
 C=A-B ' They are now subtracted

Notes : ' (single quote) and **REM** are the same

5.37 RESTORE

Syntax : **RESTORE** { *number / variable* }

Overview : Moves the pointer in a **DATA** table to the position specified by *number* or *variable*

Operators : *number* is, strangely enough, a number
variable is a user defined variable

Example : **DIM I**
 DATA 5,8,"fred",12
 RESTORE
 READ I
 ' I will now contain the value 5
 READ I
 ' I will now contain the value 8
 RESTORE 3
 ' Pointer now placed at location 4 in our data table i.e. "r"
 READ I
 ' I will now contain the value 114 i.e. the 'r' character in decimal

The data table is defined with the values 5,8,102,114,101,100,12 as "fred" equates to f:102,r:114,e:101,d:100 in decimal. The table pointer is immediately restored to the beginning of the table. This is not always required but as a general rule, it is a good idea to prevent table reading from overflowing.

The first **READ I** takes the first item of data from the table and increments the table pointer. The next **READ I** therefore takes the second item of data.

RESTORE 3 moves the table pointer to the fourth location (first location is pointer position 0) in the table - in this case where the letter 'r' is. **READ I** now retrieves the decimal equivalent of 'r' which is 114.

Notes : Alphabetic **DATA** is only allowed in LET Pic BASIC Pro. Numeric only in the LITE version.

DATA tables must be declared **after** any **DIM** statements. Attempts to read past the end of the table will result in errors and unpredictable results.

See also : **DATA , READ**

5.38 RSIN

Syntax : { *variable* } = **RSIN**

Overview : Receive a serial (RS232) byte, at inverted 9600 baud, no parity, and one stop bit 8-N-1.

Operators : *variable* is a user defined variable

Example :

```

DEVICE 16F84
INCLUDE SERIAL
DEFINE PortA=00000001 ' Configure PortA bit-0 as an input
DIM A
SYMBOL Rin=A.0 ' Assign the serial in pin to PortA.0
SYMBOL Rout=A.1 ' Assign the serial out pin to PortA.1
SYMBOL Dtr=A.2 ' Assign the handshaking pin to PortA.2
INIT SERIAL Rin,Rout,Dtr ' Inform the compiler

A=RSIN ' Read a serial byte and place in A
    
```

Notes : The baud rate of the serial byte to receive is oscillator dependant.
 If a 4MHz crystal is used, the baud rate is 9600
 If an 8MHz crystal is used, then the baud rate will double to 19200

Before the **RSIN** command may be used, it is necessary to configure the relevant port's direction, and assign the port pins: -

```

DEFINE PortA=00000001 ' Configure bit-1 of PortA as an input
RIN is the serial input pin, therefore its direction should be set to IN-
PUT
ROUT is the serial output pin, therefore its direction should be set to
OUTPUT
DTR is an optional handshaking line for use with RSIN. If it is not
used, then simply omit the SYMBOL DTR statement. However, if it is
used, then its port pin should be set as an OUTPUT.
    
```

Package : The SERIAL package must first be loaded then initialised before this command is available. Also, the two (*optional three*) port pins must be assigned.

See also : **RSOUT, INIT, INCLUDE**

5.39 RSOUT

Syntax : **RSOUT** ({ *value* })

Overview : Transmit a serial (RS232) byte, at inverted 9600 baud, no parity, and one stop bit 8-N-1.

Operators : *value* is a user defined variable or number

Example :

```

DEVICE 16F84
INCLUDE SERIAL
DEFINE PortA=00000001 ' Configure PortA bit-0 as an input
DIM A
SYMBOL Rin=A.0 ' Assign the serial in pin to PortA.0
SYMBOL Rout=A.1 ' Assign the serial out pin to PortA.1
INIT SERIAL Rin,Rout,Dtr ' Inform the compiler
' Send string '0 - Z' to rsout.-
Loop: FOR A=48 TO 90
RSOUT (A)
NEXT A
GOTO Loop
    
```

Notes : The baud rate of the serial byte to transmit is oscillator dependant.
 If a 4MHz crystal is used, the baud rate is 9600
 If an 8MHz crystal is used, then the baud rate will double to 19200

Before the **RSOUT** command may be used, it is necessary to configure the relevant port's direction, and assign the port pins: -

```

DEFINE PortA=00000001 ' Configure bit-1 of PortA as an input
RIN is the serial input pin, therefore its direction should be set to INPUT
ROUT is the serial output pin, therefore its direction should be set to OUTPUT
DTR is an optional handshaking line for use with RSIN. If it is not used, then simply omit the SYMBOL DTR statement. However, if it is used, then its port pin should be set as an OUTPUT.
    
```

Package : The SERIAL package must first be loaded then initialised before this command is available. Also, the two (*optional three*) port pins must be assigned.

See also : **RSIN, INIT, INCLUDE**

5.40 SET (or HIGH)

Syntax : **SET (or HIGH)** { *port.bit / symbol* }

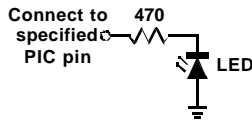
Overview : Place a port pin in the logic high position. i.e. 1

Operators : *port* can be A, B, or C
bit is the bit number to be set to 1
symbol can be a symbolic representation of a port bit

Example : **SYMBOL** LED=B.4
SET B.2
CLEAR LED
HIGH B.3
 Bit-2 of PortB is **SET** to 1. Bit-4 of PortB (symbolised by the name LED) is **CLEAR**ed to 0. Bit-3 of PortB is **SET** to 1.

Notes : There is no difference between the **SET** and **HIGH** commands.
 The pin to be used for **SET** or **HIGH** must have been previously defined as an **output**

See also : **CLEAR, DEFINE, SYMBOL**



The above diagram shows the connection of an LED to any of the pins of a PIC. A resistor must be used in series with the LED to limit the current supplied to it.

5.41 SLEEP

Syntax : **SLEEP**

Overview : Places the PIC into low power mode. i.e. power down but leaves the port pins in their previous states.

See also : **END, STOP**

5.42 SOUND

Syntax : **SOUND** ({ *pitch,length,port.bit* })

Overview : Toggles the *port bit* continuously at a speed *pitch* for a time *length*

Operators : *pitch* and *length* may be any variable or number containing values between 0 and 255
port is A, B, or C
bit is the bit number you wish to sound.

Example : **SYMBOL** Speak=B.4
 SOUND (100,100,Speak)
 SOUND (50,50,B.2)

5.43 STOP

Syntax : **STOP**

Overview : **STOP** halts program execution by sending the PIC into an indefinite loop.

Example : **IF A>12 THEN STOP**
 { *code data* }

If variable A contains a value greater than 12 stop program execution. *code data* will not be executed.

Notes : Although **STOP** halts the PIC in its tracks it does not prevent any code listed in the BASIC source after it being compiled. To do this use the **END** command.

See also : **END, SLEEP**

5.44 STORE

Syntax : **STORE** { *address* },{ *value* }

Overview : Write data into the internal eeprom of a 16C84 or 16F84

Operators : *address* is the location in the eeprom from 0-63
 value is the value to place in the eeprom

Example : **INCLUDE** EEPROM
 INIT EEPROM
 DIM A,B
 A=4 : B=10
 STORE 12,32
 STORE A,B

Put the value 32 in position 12 of the eeprom
Put the value 10 in position 4 of the eeprom

Notes : This command only applies to the 16C84 or 16F84

Package : The EEPROM module must first be loaded and initialised before the **STORE** command is available: -

INCLUDE EEPROM
INIT EEPROM

See also : **EEDATA**

5.45 SWAP

Syntax : **SWAP** { *variable 1* }, { *variable 2* }

Overview : The **SWAP** command swaps the values of two variables.

Operators : *variable 1* and *variable 2* are existing declared variables.

Example : **DIM** fred,mary
 fred=3
 mary=5
 SWAP fred,mary

fred and *mary* are our two declared variables. *fred* is given a value 3 and *mary* a value 5. After the **SWAP** command is executed, *fred* now contains 5 and *mary* contains 3.

5.46 SYMBOL

Syntax : **SYMBOL** { *name* }={ *port* },{ *bit number* }

Overview : Assign a symbolic name to represent a bit on a port.

Operators : *name* can be any symbolic name for easy use
 port can be A, B, or C
 bit number is the bit to be represented.

Example : **SYMBOL** LED=B.4
 SET LED
 CLEAR LED

Bit-4 on port B is symbolised by the name LED. This bit is **SET** to 1 and then **CLEAR**ed to 0.

Notes : **SYMBOL** must be placed in the declaration section of the program i.e. just before or after the **DIM** statement.

5.47 TIMER

Syntax : **TIMER** { *on/off* },{ 0-7 }
 { *variable* }=**TIMER**

Overview : Clears and then enables the internal timer (TMR0) of the PIC
 { *variable* }=**TIMER** to read.

Operators : *on* - clear and enable timer
 off - stop the timer
 0-7 - internal prescaler divider
 variable is a user-defined variable

Example : ‘ Flashes LED on and off
 ‘ Use RA4 as counter input pin
 ‘ LED on / off every 128 counts
 ‘ LED on port b.0
 DEVICE 16F84
 DIM A,B
 DEFINE PortB=11111110
 DEFINE PortA=11111111
 SYMBOL LED=B.0
 TIMER on,4 ‘ prescaler set to 1:32
 Loop: A=**TIMER**
 IF A>128 **THEN HIGH** LED
 IF A<128 **THEN LOW** LED
 GOTO Loop

6 - Using the PLUS version of the compiler.

The **Plus** version of the compiler allows the newer 16F87x devices to be used. These include the 16F873, 16F874, 16F876, and 16F877. All the standard commands associated with the Pro version are available, however, there are things to consider when using the new devices.

6.1 - Page boundaries

Because the 16F87x range of devices have more than 2k of flash eeprom, page boundaries come into play. This is taken care of within the compiler and is totally invisible to you, the user. For example, all GOSUBs and GOTOs will produce code that automatically adjusts the PCLATH bits.

6.2 – Analogue pins

When the 16F87x range of devices are first powered up, or reset, all pins that are capable of being used for analogue purposes are set to analogue (i.e. AN0 to AN7). This will cause problems if they are used as digital types, therefore, if your program is not using analogue inputs, these should be turned into digital types by issuing the following command at the beginning of the program: -

```
Device 16F877           ' We are using a PIC16F877
POKE (159,7)         ' Convert analogue pins to digital
```

What this does is place the value 7 into the ADCON1 register, thus disabling analogue inputs on PortA and PortE (*if applicable*).

6.3 – Using the ADIN command

The 16F87x range of devices, all contain 10-bit Analogue to Digital Converters, however, if the ADIN command is used, it will only assign the first 8 bits of the conversion into the variable. This is unavoidable because of the 8-bit nature of the compiler. But all is not lost, because the remaining 2-bits are held in the hardware register ADRESHI, which has the address 158. The Lower 8 bits are also held in the hardware register ADRESLO, which is address 30.

To access the top 4-bits, a simple peek command will suffice: -

```
ResLO=ADIN (3)       ' Get the lower 8 bits of the ADC conversion
                        ' into variable RESLO
ResHI=PEEK (158)     ' Get the top 2 bits of the ADC conversion
                        ' into variable RESHI
```


Apart from the issues noted above, the new range of PICs may be used as if they were giant 16F84s.

7 – The on-board Programmer

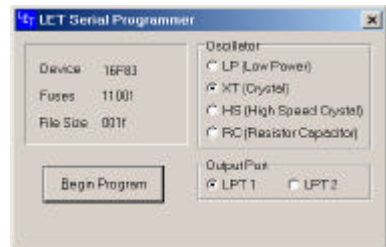
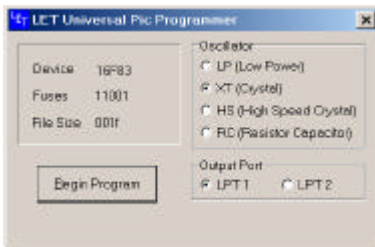
The compiler has an on-board programmer that allows quick development of your code. In fact, the compiler has support for two types of programmer, the universal (see section 8), and the In-Circuit Programmer (available end January 2001). The on-board programmers are not full implementations i.e. they are cut down programmers, for quick results.

7.1 Using the on-board Programmer

Choosing the type of programmer that you require, is accomplished by clicking on the **Compile->Setup Options->**. Click on the ISP menu to choose the In-Circuit Programmer, the default is set for the universal type.

The programmer is only available after a successful compile has been carried out. The programming  button will then become unshaded.

Depending on which type of programmer was chosen, you will be presented with the programming window.



Above, are the two programming windows for the type of programmer chosen. They look identical, apart from the text in the border, which tells you the type.

The default oscillator setting is for an XT crystal, however, this may be changed by simply choosing any of the four types available.

Note : The programmer is set for NO_WDT, which means that the PIC will not reset using the watchdog timer. If this option is required, along with any other fuse settings, you must use the full-featured programming software.

The type of PIC to program is automatically extracted from the **DEVICE** directive used in the BASIC code.

To program the PIC, simply click on the **Begin Program** button, not forgetting to choose the printer port that your programmer is attached to.

8 - Universal PIC Programmer

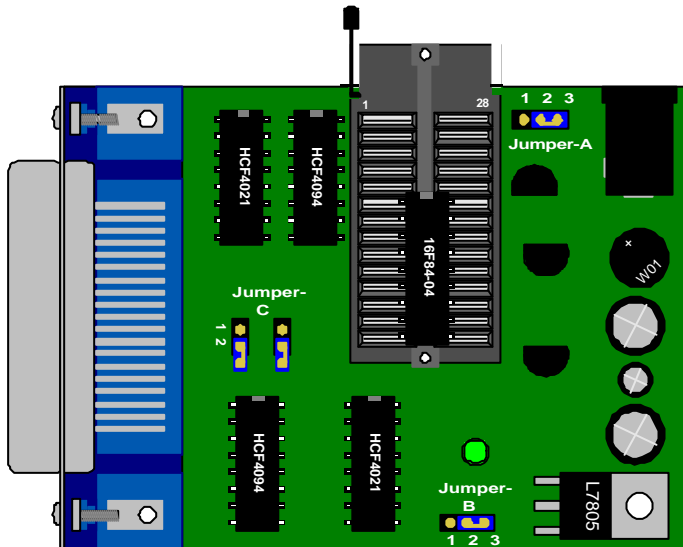
The **Universal PIC Programmer** is capable of programming many different PIC devices, as well as the 24CXX series of I²C serial eeproms. The programmer connects to the computer's parallel port (*printer port*) using a standard printer cable and is powered externally by a 12 to 18 Volt, AC or DC power supply. The use of on-board jumpers allow the selection of 18 or 28 pin devices.

Note : Because PIC devices come in all sizes and pin configurations, some devices will require adapters. These are available separately from Crownhill, or alternatively, may be constructed on stripboard. The pin arrangements for devices requiring adapters are shown in section 7.

Make sure the PIC to be programmed is removed from the ZIF socket before the programmer is powered up, as occasional initial voltage surges may damage the PIC chips.

8.1 Using the Programmer

There are four jumpers on the PIC Programmer board (*usually coloured red or blue*). These are used to configure the programmer for different device types. Their location is shown below :



**Jumper settings for programming 18-pin devices.
As well as for using the various adapters.**

8.2 - 18 pin Jumper settings

A - Middle pin connected to right pin (i.e. pins 2-3 connected)

B - Middle pin connected to right pin (i.e. pins 2-3 connected)

C - Both NOT connected (to avoid losing them I suggest placing each one over 1 pin only)

8.3 - 28 pin Jumper settings

A - Middle pin connected to left pin (i.e. pins 1-2 connected)

B - Middle pin connected to left pin (i.e. pins 1-2 connected)

C - Both connected (i.e. pins 1-2 connected)

Note : At present, only the 16C55, 16C57 and 16CR57A devices require 28-pin jumper settings. All other 28-pin devices require an adapter.

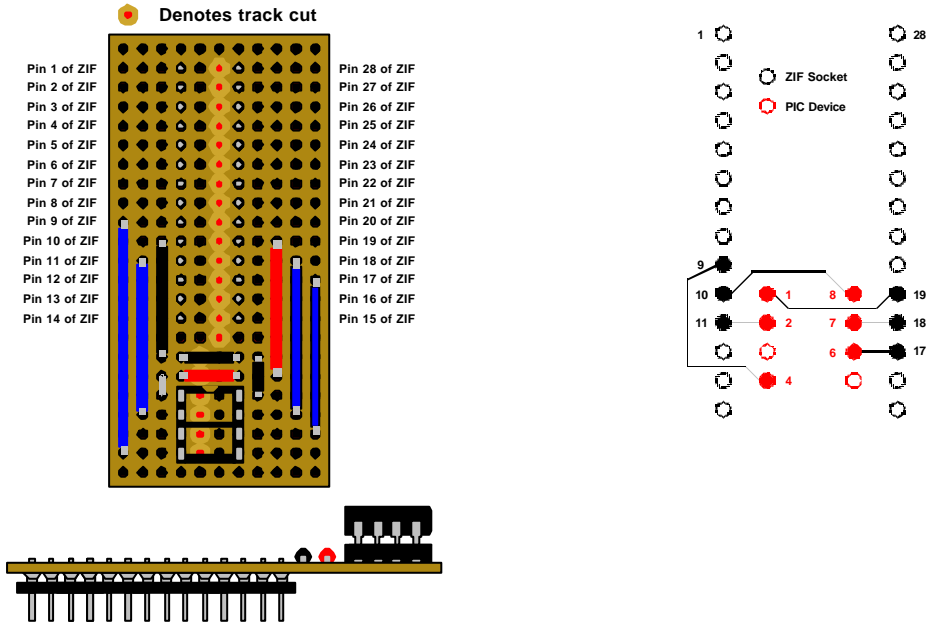
Setting the unit for 28-pin mode and then putting in a device that should be set for 18-pin may result in damage to the PIC. All devices using an adapter must be set to 18-pin.

9 - Adapter layouts

The various types of adapter are available from Crownhill, however, if you wish to produce your own, the circuit arrangements are shown as well as possible strip-board layouts for some of the more popular types of PIC.

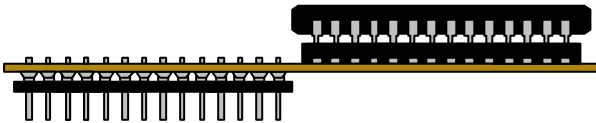
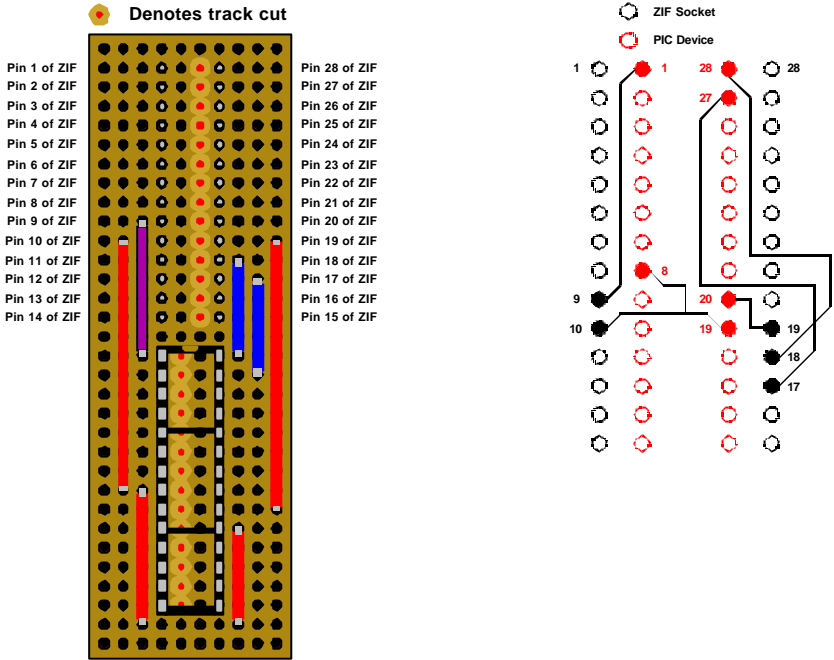
Note : When using any of the adapters, the jumpers should be configured for **18-pin** devices. Any other configuration may cause damage to the PIC.

9.1 - 8 pin Adapter. PIC types 12C508,12C509,12C671, 12C672, 12C674 etc.



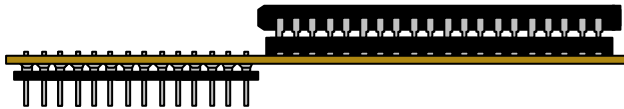
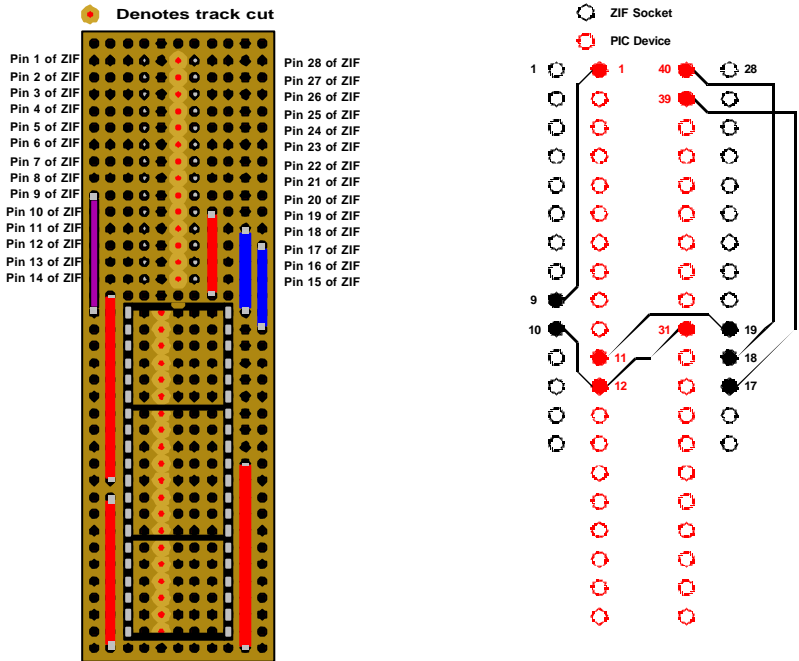
8 pin Adapter layout and circuit.

9.2 - 28 pin Adapter. PIC types 16C62 /A, 16C63, 16C66, 16C72 /A, 16C73 /A /B
16C76, 16C77X, 16F873, 16F876



28 pin adapter layout and circuit.

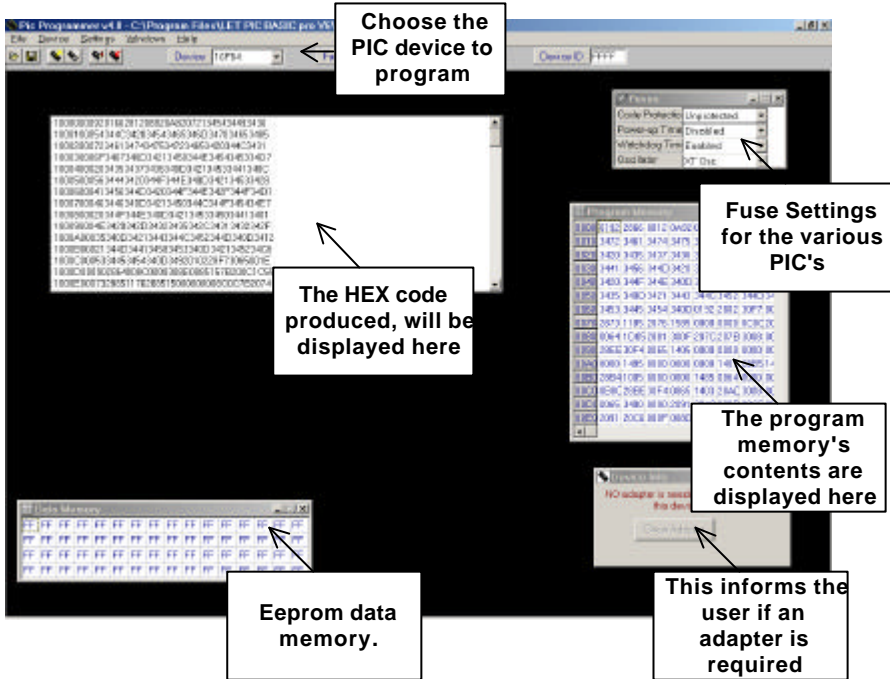
9.3 - 40 pin Adapter. PIC types 16C64 /A, 16C65 /A, 16C67, 16C74 /A /B, 16C77
16F874 , 16F877



40 pin adapter layout and circuit.

10 - Overview of the programming software

The software supplied with the programmer is pretty much self-explanatory, so we'll just briefly run through the various windows that appear on the screen.



The **Fuses** window allows you to change the configuration of the PIC's fuses, Protection on-off, Type of oscillator used etc.

The **Device Info** window informs you if an adapter is required, or not, for the particular type of PIC to be programmed.

The **Data Memory** window shows the contents of the PIC's eeprom data, if any. Not all PIC's have on-board eeprom; therefore, this window will not always appear.

The **Hex Code** and **Program Memory** windows show the contents of the PIC's main memory in 8-bit merged hex format.

10.1 - What do all the other menu options do?

Most of the menu options self-explanatory, but one needs mentioning a little further: -

The **File** menu allows you to load and save hex files.

The **Device** menu enables you to read, write, erase, or program the PIC device.

The **Windows** menu allows you to remove any or all of the separate windows i.e. fuses, device info etc.

The **Help** menu allows you to have a quick-view of adapters required for the particular device.

One of the most important menus is the **Settings** menu. This allows the programmer to be tailored to your computer. You can choose the parallel port that the programmer is connected to, and test if the programmer is operating correctly.

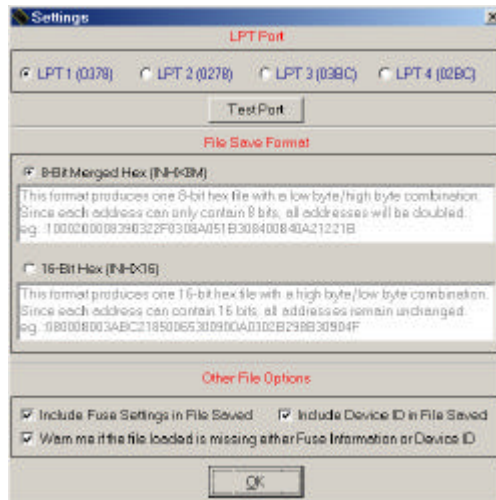
It also allows a choice of the type of hex file to program, either INHX8 or INHX16. INHX8 is the normal format, and also the type the compiler produces.

A useful feature of the programmer is the ability to save the fuse configurations that have been previously set, along with the hex file. If the hex file is reloaded, then the fuses do not require updating.

10.2 – Learning by doing

The most efficient way of learning how the programming software works is by an actual hands-on approach, therefore, we will go through the individual steps required for programming a PIC device. The PIC we will program is the ever-popular PIC16F84.

Make sure the programmer is powered up and connected to the printer port of the computer. Next, click on the **Settings** menu, and choose the printer port that your programmer is attached to, the default is LPT1. The other parameters in the settings window should look like the screen-shot below: -



Once the programmer has tested OK, we are ready to program the PIC.

Move the mouse to the **Device** window and select the PIC type to be programmed i.e. PIC16C84, PIC16F877 etc. Notice the **Device Info** window, this will inform you as to whether the PIC to be programmed requires an adapter.

Click on **File -> Open**, and choose the hex file of the program to place into the PIC. If the fuses have not been included in the assembler source code, then a warning will inform you. If the warning appeared, then move the mouse over to the **Fuses** window, and set the fuses manually. Once you are happy with all the settings, simply click on the program button, that's the one with an integrated circuit with a red arrow pointing towards it.

If all is well, then a window will appear with OK.

Simple when you know how isn't it?

Support.

Please visit the LET Basic web site <http://www.letbasic.com>, to check for updates and code examples.

Support for LET PIC BASIC is provided exclusively via the LET BASIC mailing list. Telephone support is provided on +44 (0) 1353 666709 for new users during initial installation only, thereafter all support is via the mailing list. Telephone support is NOT available for enquiries relating to syntax or other code related questions, questions of this nature should be asked via the mailing list.

Crownhill staff monitor the mailing list daily and support questions will be answered via the list. Code examples will be posted to the web page from time to time and it is our intention to build a searchable FAQ database as the product matures.

You may join the mailing list by sending an email to:

majordomo@qunos.net

with the text:

SUBSCRIBE LETBASIC-L

as the message.

You will receive a confirmation message from the mailing list manager, you must reply to the confirmation as requested, you will then be added to the mailing list. When sending messages to the mailing list you must send the message from the address that you used to subscribe to the list. Your message should be sent to:

letbasic-l@qunos.net

all list members will see your message and may reply to the message. We reserve the right not to reply to messages, or to remove messages from the list in the event of the message breaching our mailing list policy.

You may view the mailing list policy on the [letbasic.com](http://www.letbasic.com) web page.

Components at discount prices are available from <http://www.letbasic.com> web site and <http://www.crownhill.co.uk> web site.

If you do not have access to the Internet or you are unhappy with your current Internet Service provider, we recommend www.cambs.net for FREE web, email and local dial up access (local call charges apply at the time of writing) this service is provided by Crownhill Associates Ltd.

Notes.

**L.E.T
PIC BASIC
COMPILER
Version 7.0
UPDATE**

BASIC compiler for the
12C508, 12C509
16C71, 16F83, 16F84, 16F873, 16F874, 16F876, 16F877
range of PIC micro's

Please Note.

Although every precaution has been taken with the preparation of this booklet to ensure that any projects, designs or programs enclosed, operate in a correct and safe manner. The author and publisher assume no responsibility for errors or omissions. Neither is any liability assumed for the failure of any project, design or program, or any damage caused to equipment that it may be connected to, or used in combination with.

Copyright Crownhill Associates. All right reserved.

The Microchip logo and name are registered trademarks of Microchip Technologies Inc.

The L.E.T PIC BASIC Lite and Pro are registered trademarks of Crownhill Associates.

Published and distributed by Crownhill Associates Ltd
First Update Edition January 2001.

Table of Contents.

1	Introduction	3
2	What's new?	4
2.1	Assembler Code Window.	4
2.1	Built in Assembler.	5
2.3	Programmer options.	6
2.4	PRO or PLUS.	8
3	Language changes.	9
3.1	LABELS.	9
3.2	VARIABLES	10
3.3	NUMBER SYSTEM	11
3.4	ADIN	12
3.5	ASM	13
3.6	DEFINE	14
3.7	EEDATA and STORE	15

1 - Introduction

This update to the manual is necessary, to illustrate some extra features that have been incorporated into the compiler. The compiler is undergoing a gradual evolutionary process. The main difference that you'll notice (if you owned any previous versions) is the editor. No longer is it the unfriendly, and awkward to use interface that it once was, but is now a fully colour syntax highlighted, comfortable to use, and above all, friendly environment in which to write your masterpiece.

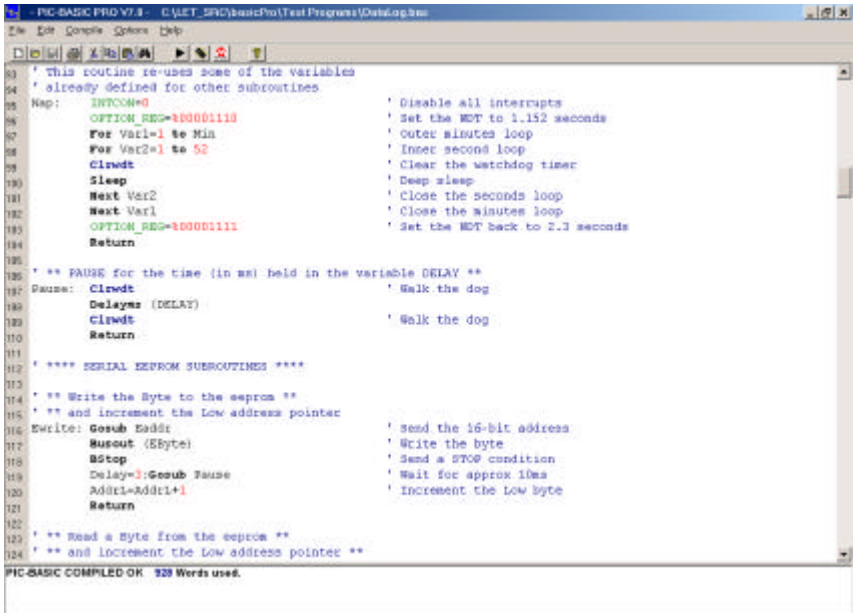
These updates, and bug fixes (if any), will be an ongoing thing. Each month or two, a slightly improved version of the compiler will be released. These updates will be in response to your feedback via the mailing list.

As it stands, the compiler is capable of producing some remarkable coding projects. But with each new release, things can only get better.

Don't worry; you won't have to learn a new dialect every time a new upgrade is brought out. Each improvement, or addition will be a natural progression from the existing structure of the language.

2 - What's new?

The facelift to the editor is the first thing we'll look at. It has several new features, as well as original features that have been improved upon.



```

53 ' This routine re-uses some of the variables
54 ' already defined for other subroutines
55 Nap:          INTCON=0          ' Disable all interrupts
56              OPTION_REG=100001110 ' Set the MDT to 1.152 seconds
57              For Var1=1 to Min  ' Outer minutes loop
58              For Var2=1 to 52    ' Inner seconds loop
59              Clwrdt             ' Clear the watchdog timer
60              Sleep             ' Deep sleep
61              Next Var2          ' Close the seconds loop
62              Next Var1         ' Close the minutes loop
63              OPTION_REG=100001111 ' Set the MDT back to 2.3 seconds
64              Return
65
66 ** FAUSE for the time (in ms) held in the variable DELAY **
67 Pause:       Clwrdt           ' Walk the dog
68              Delayms (DELAY)
69              Clwrdt           ' Walk the dog
70              Return
71
72 **** SERIAL EEPROM SUBROUTINES ****
73
74 ** Write the Byte to the eeprom **
75 ** and increment the Low address pointer
76 Write:       Gosub Eaddr      ' Send the 16-bit address
77             Busout (EByte)    ' Write the byte
78             BStop            ' Send a STOP condition
79             Delay=3:Gosub Fause ' Wait for approx 10ms
80             Addr=Addr+1      ' Increment the Low byte
81             Return
82
83 ** Read a Byte from the eeprom **
84 ** and increment the Low address pointer **
    
```

Notice the full colour syntax highlighting of keywords, numbers, hardware registers, and remarks. This makes the code extremely easy to read, and a pleasure to write.

2.1 - Assembler Code Window.

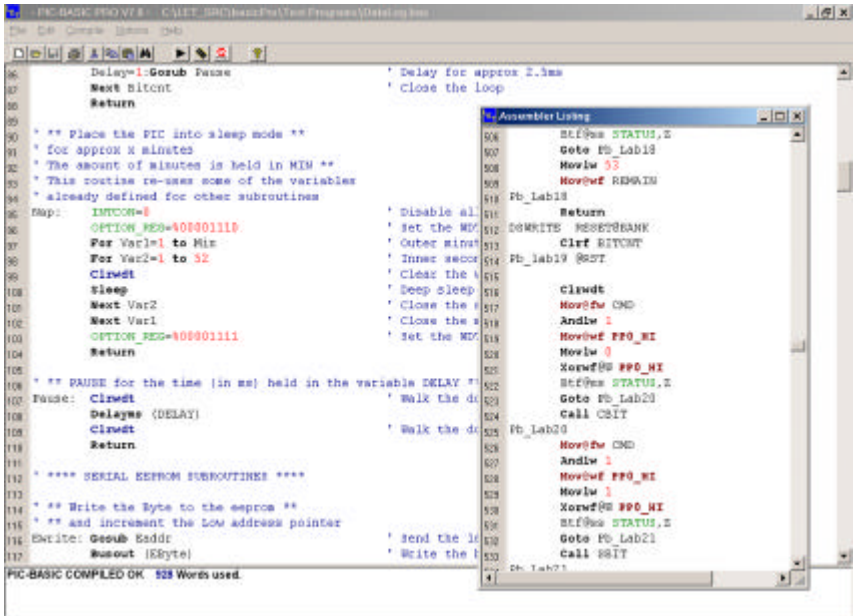
Most of the buttons have stayed the same. However, one button needs a special mention. Namely, the show assembler list button.



**Show / Hide
the
Assembler Listing**

You might be thinking that we've abandoned the window that shows the assembler code produced. This is such a useful feature that we decided to give it it's own special window that can be displayed at will. By clicking on the button with the Microchip logo, a new floating window is shown.

This also uses syntax highlighting for the assembler code, which makes any debugging that may be required, a darn sight easier.

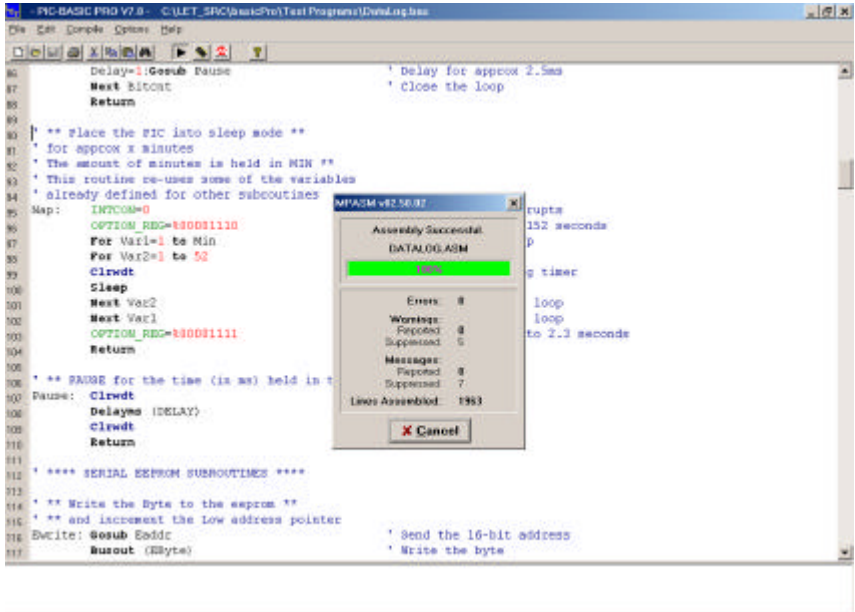


The window's dimensions and position are saved when the compiler is closed. This means that once you've found the perfect position and size for it, you needn't worry about having to change it every time the compiler is loaded.

2.2 - Built in Assembler.

The compiler now uses Microchip's™, MPASMWIN assembler. This is installed along with the compiler at setup time. This has many benefits over the previous incorporated assembler. As we'll see later.

When the compiler button, or compile menu option is selected, the program will be compiled, then MPASM will automatically assemble the code into a hex file, ready for the programmer. If any syntax errors occur within your BASIC code, then MPASM will not execute, and the error, or errors will be displayed in the error box at the bottom of the code editor. Before I forget, the error box itself is scaleable. This will enable all the errors (if applicable) to be viewed instead of a few at a time requiring scrolling.



The above screenshot is the result of a successful compile and assemble.

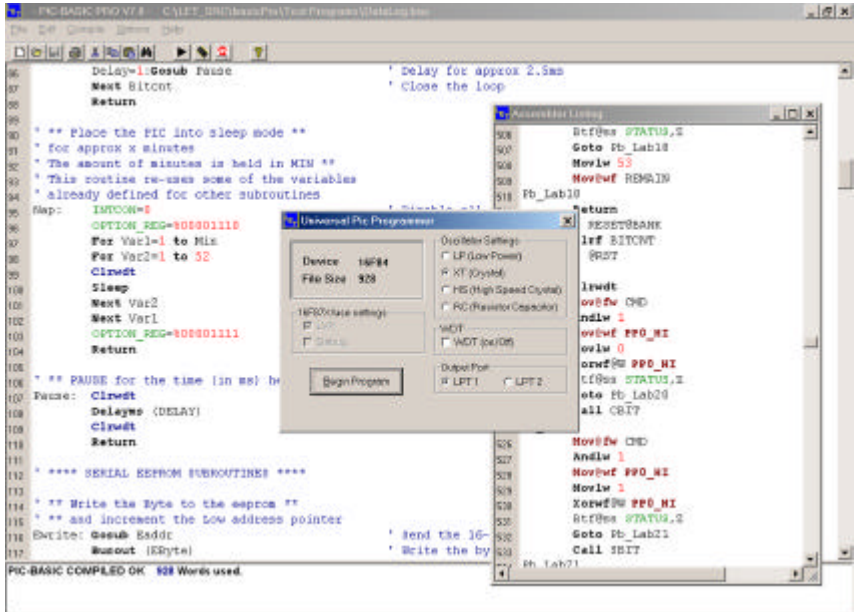
Because the compiler and assembler are two separate entities working in conjunction, there may be times when a program compiles correctly, but assemblers incorrectly. If this happens, then the assembler errors are displayed in the error window. The offending line (or lines) may then be viewed using the floating assembler window, and any corrections can be easily made.

2.3 – Programmer options.

There is now a choice of three programmers from within the confines of the editor. By choosing Compile->Programmer Options, you can choose either the Universal, In-Circuit programmer, or melab's EPIC™ programmer.



Both the Universal and the In-Circuit programming software have had improvements made to them. The programmers now take their configuration fuse settings from the hex file created by the compiler. However, you have the choice of manipulating the fuses manually when the programmer window appears.



If the EPIC option is chosen, you will be asked for the location of its working directory. This needs only to be done once, as the compiler saves the information permanently. If EPIC's working directory changes, then you can alter these details by choosing Options->Change Epic Details.



Note. that the EPIC software must already be installed on your machine. And you must own the EPIC programmer that accompanies it. The LET PIC BASIC editor does **not** have the EPIC software built in. It merely allows EPIC to be run from the compiler. The Universal, or In-Circuit programmers are **not** compatible with EPIC's software, and EPIC's hardware is **not** compatible with the Universal, or In-Circuit software.

2.4 - PRO or PLUS?

Both the PRO and PLUS versions of the compiler are now one and the same. The once separate PLUS compiler has been incorporated into the PRO version. Simply change the **DEVICE** directive to one of the 16F87x range of devices, and the compiler does the rest.

Example : **DEVICE** 16F877

All page boundary manipulation is automatic and invisible to you, the user. Virtually anything that compiles with a 16F84 or 16C71 device, will compile on one of the 16F87x devices. This has the advantage of offering, not only more memory, but by manipulating the special hardware registers associated with the new breed of PICs, hardware SPI, I2C, and UART operations may be accomplished.

Note : Because of the extra page boundary manipulation that is required when using the PICs with more than 2k of flash memory. Your code produced will be slightly longer. To overcome this to a certain degree, always place your sub-routines at the beginning of the program, not forgetting to jump over them to your main program:

Example : **DEVICE** 16F877

DIM any variables

DEFINE any ports

GOTO Main

' Subroutines are here

Sub1:

.....

Sub2

.....

' The main program starts here.

Main:

This is not required when using PICs with less than 2k of flash memory. But it's a good habit to get into anyway.

3 - Language changes.

This upgrade to version 7.0 of the compiler is not only cosmetic. Improvements have been made to the actual BASIC language. These will be outlined in the following section.

3.1 – LABELS.

LABELs may now have numerical content, and an underscore. This makes for much more understandable code. Consider: -

LABELTWO:

and

LABEL2:

Each has its merits, but the second version is not only easier to write, but is easier to spot in a large program.

Consider also: -

LABELTWO:

and

LABEL_TWO:

Again, the second version is easier to understand and spot in a large program.

Note : Underscores and numbers may be freely mixed, but a label cannot start with a number, or the compiler will issue an error: -

2LABEL: is NOT allowed.

Labels can be up to 32 characters in length.

3.2 - VARIABLES.

Although, still 8-bit in nature, variables have had a major overhaul.

Variable names, as in the case of labels, may now freely mix numeric content and underscores.

Example : **DIM** MyVar
 or
 DIM My_Var
 or
 DIM My_Var2

Variable names may start with an underscore, but must not start with a number. Variables may also be up to 32 characters in length.

DIM 2MyVar is NOT allowed.

Variable assignment has also been improved. The compiler now recognises all the hardware registers associated with the PIC chosen by the **DEVICE** directive.

Example : **INTCON=1** ‘ Will load the INCON register with value 1

This in fact eliminates the need for the commands **PEEK** and **POKE**, because the variable can now be read or written too directly.

Instead of using: -

MyVar=**PEEK**(3) ‘ Read the contents of the STATUS register

You can use: -

MyVar=**STATUS**

And...

POKE (3,2) ‘ Load the STATUS register with value 2

You can use: -

STATUS=2

This also eliminates the use of the **OUTA,B,C,D** command and the **IN-PORTA,B,C,D** command, as the variable's name may be used in an **IF-THEN** statement:-

IF PORTA=2 **THEN** do something

3.3 - NUMBER SYSTEMS.

The compiler now recognises Hexadecimal, Binary, and Decimal numeric types.

A **HEXADECIMAL** number is preceded by the \$ character: -

MyVar = \$10 ' Load MyVar with a value of 16

A **BINARY** number is preceded by the % character: -

MyVar = %10000000 ' Load MyVar with a value of 128

Binary numbers do not need to be 8 characters in length.

The value %1001

Is the same as

%00001001

A **DECIMAL** number is not preceded by any character.

Binary notation has the advantage when setting a particular sequence of bits in a register. Consider:-

INTCON = 9 ' Set bits 0 and 3 of INTCON

And

INTCON = %1001 ' Set bits 0 and 3 of INTCON

Binary also comes into its own when using the bit wise AND, OR, XOR operators. Because you can plainly see which bits are being masked or set: -

MyVar = MyVar & %00001111 ' Mask the lower 4 bits of MyVar

3.4 - ADIN

The **ADIN** command now works with any of the devices that contain an on-board ADC. The operation of the **ADIN** command has been slightly altered, in that, the **ADCON1** register needs to be manipulated before the **ADIN** command is issued.

The **ADCON1** register configures the port pins for either analogue input or digital. Consult the PIC's datasheet for the value to place into **ADCON1** for a particular configuration.

And also, before the **ADIN** command is used, the pin/s of interest must be configured as inputs, either by using the **DEFINE** command or setting the port's TRIS value i.e. **TRISA** = number.

channel number may now contain either a user defined variable or numeric value between 0 and the amount of ADC channels available on the selected device

Example : ' Retrieve the value of channel 3
 ' of the A to D Converter and place into variable MyVar.

```

DEVICE 16F877
INCLUDE A2D
INIT A2D
DIM MyVar
DEFINE PortA %00001000 ' Configure AN3 (PortA.3) as an input
INTCON1= %100            ' Set AN3 as analogue
MyVar=ADIN (3)            ' Place the conversion into variable MyVar
    
```

The 16F87x range of devices, all contain 10-bit Analogue to Digital Converters, however, if the **ADIN** command is used, it will only assign the first 8 bits of the conversion into the variable. This is unavoidable because of the 8-bit nature of the compiler. But all is not lost, because the remaining 2-bits are held in the hardware register **ADRESH**. The Lower 8 bits are also held in the hardware register **ADRESL**.

To access the top 2-bits, use: -

```

ResLO=ADIN (3)            ' Get the lower 8 bits of the ADC conversion
                         ' into variable RESLO

ResHI=ADRESL            ' Get the top 2 bits of the ADC conversion
                         ' into variable RESHI
    
```

3.5 - ASM

Because the compiler now uses a separate assembler, the **ASM** directive has had a major overhaul. There are now three ways of using assembler in the compiler.

The ASM directive no longer uses the curly braces, but uses the format: -

ASM

Mnemonics

ENDASM

Any mnemonics used within these two directives, are passed directly to MPASM, without the compiler's intervention. This allows FULL control over the PIC in use. It also allows the extra features of MPASM to be used, such as Macros, Conditional assembly etc.

Another way of passing mnemonics directly to MPASM is by using the @ character in front of the mnemonic: -

```
@    MOVLW 100
```

The third way of using assembler in the compiler is in the form of extra commands. All the mnemonics may now be used freely along with BASIC commands. Take the example: -

```
DIM MyVar
MyVar=10           ' Load MyVar with a value of 10
BSF MyVar,1      ' Set bit 1 of MyVar
PRINT MyVar," "  ' Print the value of MyVar
```

If mnemonics are used alongside BASIC commands, they are treated nearly the same as BASIC commands, and must obey the same rules, such as, the variable must be declared before it is used.

If there is a problem with your code, assembler errors will be produced instead of compiler errors, if the mnemonics are passed directly to MPASM.

The ability to freely mix BASIC and mnemonics is an incredibly powerful tool. But it must be used wisely.

3.6 - DEFINE

With the new numbering system now in place, the **DEFINE** command's syntax has been slightly altered to allow either Hex, Binary, or Decimal values to configure the Port bits: -

Instead of using the previous: -

```
DEFINE PortX = 10001010
```

A more flexible approach has been adopted, in that, the *equals* assignment is now optional, and the numeric value does not need to be a binary value containing 8 characters. For example: -

```
DEFINE PortA %1001
```

will configure PortA, bits 0 and 3 as inputs. And so will....

```
DEFINE PortA = %1001
```

Another use for the **DEFINE** command is to set general parameters for the compiler. This will be used extensively in future upgrades, but for now, only one has been implemented: -

```
DEFINE REMARKS 0
```

and...

```
DEFINE REMARKS 1
```

This command removes the commented BASIC statements from the assembler code, thus reducing its size considerably. It does not affect the compiled code in any way. But if you have a large program with many remarks, MPASM might complain about the size of the .ASM file.

3.7 - EEDATA and STORE

Both these commands now work with any of the devices that support on-board eeprom. Their syntax has not changed in any way, and their operation is invisible to the user. But you must remember that each PIC type has differing amounts of on-board eeprom.

Example 1 : **DEVICE** 16F84
 DIM My_Var
 STORE 24,127
 MyVar=**EEDATA** (24)
 PRINT MyVar," "

The above program will write and read address 24 of the on-board eeprom, of the 16F84 device.

Example 2 : **DEVICE** 16F874
 DIM My_Var
 STORE 24,127
 MyVar=**EEDATA** (24)
 PRINT MyVar," "

The above program will do the exact same thing as example 1, except on a 16F874 device.