

Crownhill reserves the right to make changes to the products contained in this publication in order to improve design, performance or reliability. Except for the limited warranty covering a physical CD-ROM and/or Hardware License key supplied with this publication as provided in the End-User License agreement, the information and material content of this publication and possible accompanying CD-ROM are provided “as is” without warranty of any kind express or implied including without limitation any warranty concerning the accuracy adequacy or completeness of such information or material or the results to be obtained from using such information or material. Neither Crownhill Associates Limited or the author shall be responsible for any claims attributable to errors omissions or other inaccuracies in the information or materials contained in this publication and in no event shall Crownhill Associates or the author be liable for direct indirect or special incidental or consequential damages arising out of the use of such information or material. Neither Crownhill or the author convey any license under any patent or other right, and make 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.

All terms mentioned in this manual that are known to be trademarks or service marks have been appropriately marked. Use of a term in this publication should not be regarded as affecting the validity of any trademark.

PICmicro™ is a trade name of Microchip Technologies Inc. www.microchip.com

Proton™ is a trade name of Crownhill Associates Ltd. www.crownhill.co.uk

Proton24™ is a trade name of Crownhill Associates Ltd. www.crownhill.co.uk

EPIC™ is a trade name of microEngineering Labs Inc. www.microengineeringlabs.com

The Proton IDE was written by David Barker of Mecanique www.mecanique.co.uk

Proteus VSM © Copyright Labcenter Electronics Ltd www.labcenter.co.uk

Web url’s correct at time of publication.

The Proton compiler and documentation was written by Les Johnson

If you should find any anomalies or omission in this document, please contact us, as we appreciate your assistance in improving our products and services.

First published by Crownhill Associates Limited, Cambridge, England, 2004.

Introduction

The Proton compiler was written with simplicity and flexibility in mind. Using BASIC, which is almost certainly the easiest programming language around, you can now produce extremely powerful applications for your PICmicro™ without having to learn the relative complexity of assembler, or wade through the gibberish that is C.

The Proton IDE provides a seamless development environment, which allows you to write, debug and compile your code within the same Windows environment, and by using a compatible programmer, just one key press allows you to program and verify the resulting code in the PICmicro™ of your choice!

Contact Details

For your convenience we have set up a web site **www.picbasic.org**, where there is a section for users of the Proton compiler, to discuss the compiler, and provide self help with programs written for Proton 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.

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

Postal

Crownhill Associates Limited.
Old Station Yard
Station Road
Ely
Cambridgeshire.
CB6 3PZ.

Telephone

(+44) 01353 749990

Fax

(+44) 01353 749991

Email

sales@crownhill.co.uk

Web Sites

www.crownhill.co.uk
www.protonbasic.co.uk

Table of Contents.

Proton IDE Overview	11
Menu Bar	12
Main Toolbar	13
Edit Toolbar	14
Code Explorer	16
Results View	19
Editor Options	20
Highlighter Options.....	22
Compile and Program Options	24
Installing a Programmer.....	25
Creating a custom Programmer Entry.....	26
Microcode Loader	28
Loader Options.....	30
Loader Main Toolbar	31
IDE Plugins	32
ASCII Table	33
Hex View	33
Assembler Window	34
Assembler Main Toolbar	35
Assembler Editor Options	36
Serial Communicator.....	37
Serial Communicator Main Toolbar.....	38
Labcenter Electronics Proteus VSM.....	41
ISIS Simulator Quick Start Guide	41
Compiler Overview.....	44
PICmicro™ Devices.....	45
Limited 12-bit Device Compatibility.	45
Programming Considerations for 12-bit core Devices.	46
Device Specific issues	47
Identifiers	48
Line Labels	48
Variables	49

Floating Point Math.....	52
Floating Point To Integer Rounding	54
Floating Point Exception Flags.....	55
Aliases.....	56
Constants	59
Symbols	59
Numeric Representations	60
Quoted String of Characters	60
Ports and other Registers.....	60
General Format	61
A Typical basic Program Layout	62
Line Continuation Character '_'	63
Creating and using Arrays	64
Creating and using Strings	71
Creating and using Code Memory Strings	77
Creating and using Eeprom Memory Strings with Edata	79
String Comparisons	81
Relational Operators	84
Boolean Logic Operators	85
Math Operators	86
Add.....	87
Subtract	87
Multiply	88
Multiply High.....	89
Multiply Middle	89
Divide.....	90
Integer Modulus.....	91
Logical and	92
Logical or.....	92
Logical Xor.....	93
Bitwise Shift Left	93
Bitwise Shift Right	94
Complement	94
Bitwise Reverse '@'	94
Decimal Digit extract '?'	94
Abs	95

fAbs	96
Acos	97
Asin	98
Atan	99
Cos	100
Dcd	101
Exp	102
fRound	103
ISin	104
ICos	105
Isqr	106
Log	107
Log10	108
Ncd	109
Pow	110
Sin	111
Sqr	112
Tan	113
Div32	114
Commands and Directives	115
Adin	119
Asm..EndAsm	121
Box	122
Branch	123
BranchL	124
Break	125
Bstart	127
Bstop	128
Brestart	128
BusAck	128
BusNack	128
Busin	129
Busout	132
Button	136
Call	138

Cdata	139
CF_Init	144
CF_Sector	145
CF_Read	150
CF_Write	153
Circle.....	157
Clear	158
ClearBit	159
Clc	160
Config	161
Config1 and Config2.....	162
Continue.....	163
Context	164
Context Save.....	164
Context Restore.....	164
Counter	166
cPtr8, cPtr16, cPtr32.....	167
Cread	169
Cread8, Cread16, Cread32	170
Cursor	172
Cwrite	173
Dec	174
Declare.....	175
Oscillator Frequency Declare.....	175
Misc Declares.	176
Adin Declares.	180
Busin - Busout Declares.....	180
Hbusin - Hbusout Declare.....	181
Hserin, Hserout, Hrsin and Hrsout Declares.....	181
USART2 Declares for use with Hrsin2, Hserin2, Hrsout2 and Hserout2.....	182
Hpwmm Declares.	183
Alphanumeric (Hitachi) LCD Print Declares.	184
Graphic LCD Declares.....	185
Samsung KS0108 Graphic LCD specific Declares.	185
Toshiba T6963 Graphic LCD specific Declares.	186
Keypad Declare.....	188
Rsin - Rsout Declares.....	189
Serin - Serout Declare.....	190
Shin - Shout Declare.....	191
Compact Flash Interface Declares.....	191
DelayCs	194
DelayMs	195
DelayUs.....	196
Device	197

Dig.....	198
Dim.....	199
Disable.....	204
DTMFout.....	205
Edata.....	206
Enable.....	211
Software Interrupts in BASIC.....	212
End.....	213
Eread.....	214
Ewrite.....	215
For...Next...Step.....	216
Freqout.....	218
GetBit.....	220
Gosub.....	221
Goto.....	225
HbStart.....	226
HbStop.....	227
HbRestart.....	227
HbusAck.....	227
HbusNack.....	227
Hbusin.....	228
Hbusout.....	231
High.....	234
Hpwm.....	235
Hrsin.....	236
Hrsout.....	242
Hserin.....	247
Hserout.....	253
I2Cin.....	258
I2Cout.....	260
If..Then..ElseIf..Else..EndIf.....	263
Include.....	265
Inc.....	267
Inkey.....	268
Input.....	269
LCDread.....	270

LCDwrite	272
Ldata	274
Len	279
Left\$	281
Line.....	283
LineTo.....	284
LoadBit.....	285
LookDown	286
LookDownL.....	287
LookUp.....	288
LookUpL	289
Low.....	290
Lread	291
Lread8, Lread16, Lread32	294
Mid\$.....	296
On Goto	298
On GotoL.....	300
On Gosub	301
On_Hardware_Interrupt.....	303
Typical format of the interrupt handler with standard 14-bit core devices.....	304
Typical format of the interrupt handler with enhanced 14-bit core devices.	304
Typical format of the interrupt handler with 18F devices.	305
On_Low_Interrupt	306
Output	309
Org	310
Oread.....	311
Owrite	316
Pixel.....	318
Plot	319
Pop	321
Pot.....	323
Print.....	324
Using a Samsung KS0108 Graphic LCD	329
Using a Toshiba T6963 Graphic LCD	334
Ptr8, Ptr16, Ptr32	337
PulseIn.....	340
PulseOut.....	341
Push.....	342

Pwm.....	347
Random.....	348
RC5in	349
RCin	350
Repeat...Until.....	353
Resume	354
Return	355
Right\$	357
Rsin	359
Rsout	364
Seed	369
Select..Case..EndSelect	370
Serin	372
Serout	379
Servo	387
SetBit	389
Set_OSCCAL	390
Set	391
Shin	392
Shout	394
Snooze	396
Sleep.....	397
SonyIn	399
Sound	400
Sound2.....	401
Stop	402
Strn.....	403
Str\$	404
Swap.....	406
Symbol	407
Toggle.....	408
ToLower	409
ToUpper	411
Toshiba_Command	413
Toshiba_UDG	417
UnPlot	419

USBinit	420
USBin	421
USBout	423
USBpoll	427
Val	428
AddressOf or VarPtr	430
While...Wend	431
Xin	432
Xout	434
Using the Optimiser	436
Caveats	437
Using the Preprocessor	438
Preprocessor Directives	438
Conditional Directives (\$ifdef, \$ifndef, \$if, \$endif, \$else and \$elseif)	441
Using the Proton Compiler with MPLAB IDE™	444
Protected Compiler Words	453

Proton IDE Overview

Proton IDE is a professional and powerful Integrated Development Environment (IDE) designed specifically for the Proton compiler. Proton IDE is designed to accelerate product development in a comfortable user friendly environment without compromising performance, flexibility or control.

Code Explorer

Possibly the most advanced code explorer for PICmicro™ based development on the market. Quickly navigate your program code and device Special Function Registers (SFRs).

Compiler Results

Provides information about the device used, the amount of code and data used, the version number of the project and also date and time. You can also use the results window to jump to compilation errors.

Programmer Integration

The Proton IDE enables you to start your preferred programming software from within the development environment . This enables you to compile and then program your microcontroller with just a few mouse clicks (or keyboard strokes, if you prefer).

Integrated Bootloader

Quickly download a program into your microcontroller without the need of a hardware programmer. Bootloading can be performed in-circuit via a serial cable connected to your PC.

Real Time Simulation Support

Proteus Virtual System Modelling (VSM) combines mixed mode SPICE circuit simulation, animated components and microprocessor models to facilitate co-simulation of complete microcontroller based designs. For the first time ever, it is possible to develop and test such designs before a physical prototype is constructed.

Serial Communicator

A simple to use utility which enables you to transmit and receive data via a serial cable connected to your PC and development board. The easy to use configuration window allows you to select port number, baudrate, parity, byte size and number of stop bits. Alternatively, you can use Serial Communicator favourites to quickly load pre-configured connection settings.

Online Updating

Online updates enable you to keep right up to date with the latest IDE features and fixes.

Plugin Architecture

The Proton IDE has been designed with flexibility in mind with support for IDE plugins.

Supported Operating Systems

Windows XP 32-bit or Windows 7 32-bit or 64-bit

Minimum Hardware Requirements

1 GHz Processor
1 GB RAM
40 GB hard drive space
16 bit graphics card.

Menu Bar

File Menu

- **New** - Creates a new document. A header is automatically generated, showing information such as author, copyright and date. To toggle this feature on or off, or edit the header properties, you should select editor options.
- **Open** - Displays a open dialog box, enabling you to load a document into the Proton IDE. If the document is already open, then the document is made the active editor page.
- **Save** - Saves a document to disk. This button is normally disabled unless the document has been changed. If the document is 'untitled', a save as dialog is invoked. A save as dialog is also invoked if the document you are trying to save is marked as read only.
- **Save As** - Displays a save as dialog, enabling you to name and save a document to disk.
- **Close** - Closes the currently active document.
- **Close All** - Closes all editor documents and then creates a new editor document.
- **Reopen** - Displays a list of Most Recently Used (MRU) documents.
- **Print Setup** - Displays a print setup dialog.
- **Print Preview** - Displays a print preview window.
- **Print** - Prints the currently active editor page.
- **Exit** - Enables you to exit the Proton IDE.

Edit Menu

- **Undo** - Cancels any changes made to the currently active document page.
- **Redo** - Reverse an undo command.
- **Cut** - Cuts any selected text from the active document page and places it into the clipboard. This option is disabled if no text has been selected. Clipboard data is placed as both plain text and RTF.
- **Copy** - Copies any selected text from the active document page and places it into the clipboard. This option is disabled if no text has been selected. Clipboard data is placed as both plain text and RTF.
- **Paste** - Paste the contents of the clipboard into the active document page. This option is disabled if the clipboard does not contain any suitable text.
- **Delete** - Deletes any selected text. This option is disabled if no text has been selected.
- **Select All** - Selects the entire text in the active document page.
- **Change Case** - Allows you to change the case of a selected block of text.

- **Find** - Displays a find dialog.
- **Replace** - Displays a find and replace dialog.
- **Find Next** - Automatically searches for the next occurrence of a word. If no search word has been selected, then the word at the current cursor position is used. You can also select a whole phrase to be used as a search term. If the editor is still unable to identify a search word, a find dialog is displayed.

View Menu

- **Results** - Display or hide the results window.
- **Code Explorer** - Display or hide the code explorer window.
- **Loader** - Displays the MicroCode Loader application.
- **Loader Options** - Displays the MicroCode Loader options dialog.
- **Compile and Program Options** - Displays the compile and program options dialog.
- **Editor Options** - Displays the application editor options dialog.
- **Toolbars** - Display or hide the main, edit and compile and program toolbars. You can also toggle the toolbar icon size.
- **Plugin** - Display a drop down list of available IDE plugins.

Help Menu

- **Help Topics** - Displays the helpfile section for the toolbar.
- **Online Forum** - Opens your default web browser and connects to the online Proton Plus developer forum.
- **About** - Display about dialog, giving both the Proton IDE and Proton compiler version numbers.

Main Toolbar



New

Creates a new document. A header is automatically generated, showing information such as author, copyright and date. To toggle this feature on or off, or edit the header properties, you should select the editor options dialog from the main menu.



Open

Displays a open dialog box, enabling you to load a document into the Proton IDE. If the document is already open, then the document is made the active editor page.



Save

Saves a document to disk. This button is normally disabled unless the document has been changed. If the document is 'untitled', a save as dialog is invoked. A save as dialog is also invoked if the document you are trying to save is marked as read only.



Cut

Cuts any selected text from the active document page and places it into the clipboard. This option is disabled if no text has been selected. Clipboard data is placed as both plain text and RTF.



Copy

Copies any selected text from the active document page and places it into the clipboard. This option is disabled if no text has been selected. Clipboard data is placed as both plain text and RTF.



Paste

Paste the contents of the clipboard into the active document page. This option is disabled if the clipboard does not contain any suitable text.



Undo

Cancels any changes made to the currently active document page.



Redo

Reverse an undo command.



Print

Prints the currently active editor page.

Edit Toolbar



Find

Displays a find dialog.



Find and Replace

Displays a find and replace dialog.



Indent

Shifts all selected lines to the next tab stop. If multiple lines are not selected, a single line is moved from the current cursor position. All lines in the selection (or cursor position) are moved the same number of spaces to retain the same relative indentation within the selected block. You can change the tab width from the editor options dialog.



Outdent

Shifts all selected lines to the previous tab stop. If multiple lines are not selected, a single line is moved from the current cursor position. All lines in the selection (or cursor position) are moved the same number of spaces to retain the same relative indentation within the selected block. You can change the tab width from the editor options dialog.



Block Comment

Adds the comment character to each line of a selected block of text. If multiple lines are not selected, a single comment is added to the start of the line containing the cursor.



Block Uncomment

Removes the comment character from each line of a selected block of text. If multiple lines are not selected, a single comment is removed from the start of the line containing the cursor.

Compile and Program Toolbar



Compile

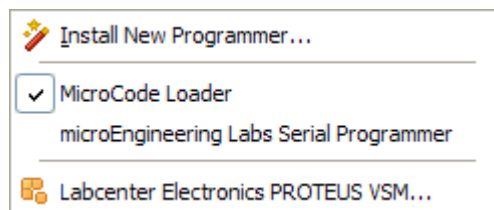
Pressing this button, or F9, will compile the currently active editor page. The compile button will generate a *.hex file, which you then have to manually program into your microcontroller. Pressing the compile button will automatically save all open files to disk. This is to ensure that the compiler is passed an up to date copy of the file(s) your are editing.



Compile and Program

Pressing this button, or F10, will compile the currently active editor page. Pressing the compile and program button will automatically save all open files to disk. This is to ensure that the compiler is passed an up to date copy of the file(s) your are editing.

Unlike the compile button, the Proton IDE will then automatically invoke a user selectable application and pass the compiler output to it. The target application is normally a device programmer, for example, MicroCode Loader. This enables you to program the generated *.hex file into your MCU. Alternatively, the compiler output can be sent to an IDE Plugin. For example, the Labcenter Electronics Proteus VSM simulator. You can select a different programmer or Plugin by pressing the small down arrow, located to the right of the compile and program button...



In the above example, MicroCode Loader has been selected as the default device programmer. The compile and program drop down menu also enables you to install new programming software. Just select the 'Install New Programmer...' option to invoke the programmer configuration wizard. Once a program has been compiled, you can use F11 to automatically start your programming software or plugin. You do not have to re-compile, unless of course your program has been changed.



Loader Verify

This button will verify a *.hex file (if one is available) against the program resident on the microcontroller. The loader verify button is only enabled if MicroCode Loader is the currently selected programmer.



Loader Read

This button will upload the code and data contents of a microcontroller to MicroCode Loader. The loader read button is only enabled if MicroCode Loader is the currently selected programmer.



Loader Erase

This button will erase program memory for the 18Fxxx(x) series of microcontroller. The loader erase button is only enabled if MicroCode Loader is the currently selected programmer.



Loader Information

This button will display the microcontroller loader firmware version. The loader information button is only enabled if MicroCode Loader is the currently selected programmer.

Code Explorer

The code explorer enables you to easily navigate your program code. The code explorer tree displays your currently selected processor, include files, declares, constants, variables, alias and modifiers, labels, macros and data labels.


Device Node

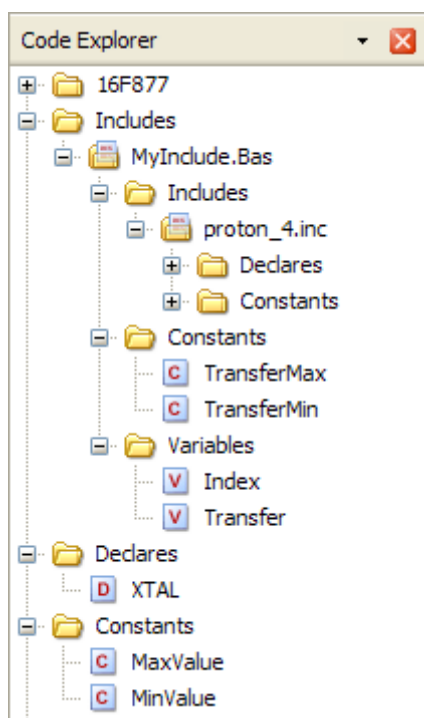
The device node is the first node in the explorer tree. It displays your currently selected processor type. For example, if your program has the declaration: -



```
Device = 16F877
```

then the name of the device node will be 16F877. You don't need to explicitly give the device name in your program for it to be displayed in the explorer. For example, you may have an include file with the device type already declared. The code explorer looks at all include files to determine the device type. The last device declaration encountered is the one used in the explorer window. If you expand the device node, then all Special Function Registers (SFRs) belonging to the selected device are displayed in the explorer tree.

Include File Node

When you click on an include file, the IDE will automatically open that file for viewing and editing. Alternatively, you can just explore the contents of the include file without having to open it. To do this, just click on the  icon and expand the node. For example: -



In the above example, clicking on the  icon for MyInclude.bas has expanded the node to reveal its contents. You can now see that MyInclude.bas has two constant declarations called TransferMax and TransferMin and also two variables called Index and Transfer. The include file also contains another include file called proton_4.inc. Again, by clicking the  icon, the contents of proton_4.inc can be seen, without opening the file. Clicking on a declaration name will open the include file and automatically jump to the line number. For example, if you were to click on TransferMax, the include file MyInclude.bas would be opened and the declaration TransferMax would be marked in the IDE editor window.

When using the code explorer with include files, you can use the explorer history buttons to go backwards or forwards. The explorer history buttons are normally located to the left of the main editors file select tabs,


- ← History back button
- History forward button

Additional Nodes

Declares, constants, variables, alias and modifiers, labels, macros and data label explorer nodes work in much the same way. Clicking on any of these nodes will take you to its declaration. If you want to find the next occurrence of a declaration, you should enable automatically select variable on code explorer click from *View...Editor Options*.

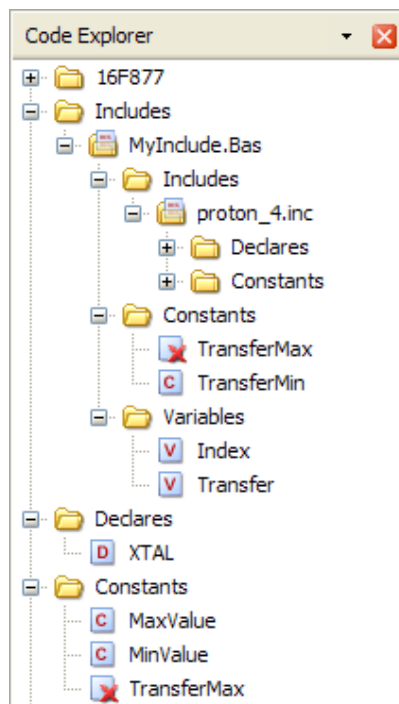
Selecting this option will load the search name into the 'find dialog' search buffer. You then just need to press F3 to search for the next occurrence of the declaration in your program. To sort the explorer nodes, right click on the code explorer and check the Sort Nodes option.

Explorer Warnings and Errors


The code explorer can identify duplicate declarations. If a declaration duplicate is found, the explorer node icon changes from its default state to a . For example,

```
Dim MyVar as Byte
Dim MyVar as Byte
```

The above example is rather simplistic. It is more likely you see the duplicate declaration error in your program without an obvious duplicate partner. That is, only one single duplicate error symbol is being displayed in the code explorer. In this case, the declaration will have a duplicate contained in an include file. For example,



The declaration TransferMax has been made in the main program and marked as a duplicate. By exploring your include files, the problem can be identified. In this example, TransferMax has already been declared in the include file MyInclude.bas

Some features of the compiler are not available for some MCU types. For example, you cannot have a string declaration when using a 14 core part (for example, the 16F877). If you try to do this, the explorer node icon changes from its default state and displays a . You will also see this icon displayed if the SFR View feature for a device is not available.

Notes

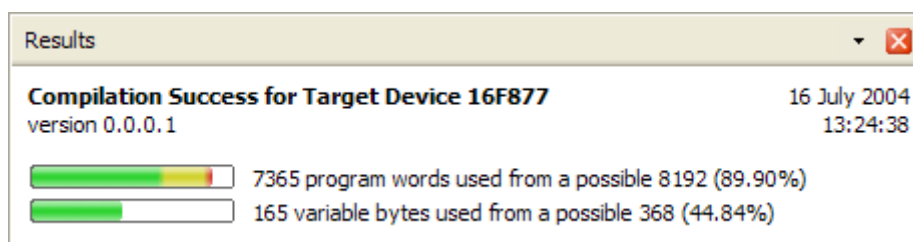
The code explorer uses an optimised parse and pattern match strategy in order to update the tree in real time. The explorer process is threaded so as not to interfere or slow down other IDE tasks, such as typing in new code. However, if you run computationally expensive background tasks on your machine (for example, circuit simulation) you will notice a drop in update performance, due to the threaded nature of the code explorer.

Results View

The results view performs two main tasks. These are (a) display a list of error messages, should either compilation or assembly fail and (b) provide a summary on compilation success.

Compilation Success View

By default, a successful compile will display the results success view. This provides information about the device used, the amount of code and data used, the version number of the project and also date and time.



If you don't want to see full summary information after a successful compile, select *View...Editor Options* from the IDE main menu and uncheck display full summary after successful compile. The number of program words (or bytes used, if its a 16 core device) and the number of data bytes used will still be displayed in the IDE status bar.

Version Numbers

The version number is automatically incremented after a successful build. Version numbers are displayed as major, minor, release and build. Each number will rollover if it reaches 256. For example, if your version number is 1.0.0.255 and you compile again, the number displayed will be 1.0.1.0. You might want to start your version information at a particular number. For example 1.0.0.0. To do this, click on the version number in the results window to invoke the version information dialog. You can then set the version number to any start value. Automatic incrementing will then start from the number you have specified. To disable version numbering, click on the version number in the results window to invoke the version information dialog and then uncheck enable version information.

Date and Time

Date and time information is extracted from the generated *.hex file and is always displayed in the results view.

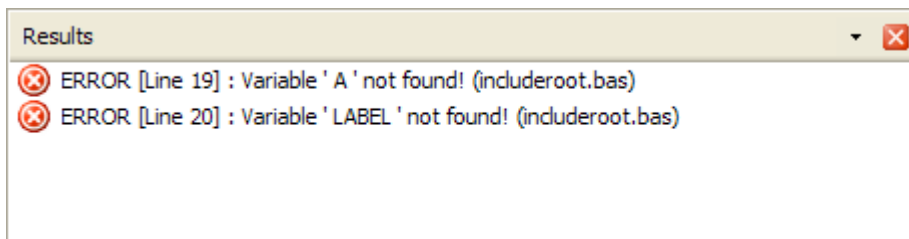
Success - With Warnings!

A compile is considered successful if it generates a *.hex file. However, you may have generated a number of warning messages during compilation. Because you should not normally ignore warning messages, the IDE will always display the error view, rather than the success view, if warnings have been generated.

To toggle between these different views, you can do one of the following click anywhere on the IDE status bar right click on the results window and select the Toggle View option.

Compilation Error View

If your program generates warning or error messages, the error view is always displayed.



Clicking on each error or warning message will automatically highlight the offending line in the main editor window. If the error or warning has occurred in an include file, the file will be opened and the line highlighted. By default, the IDE will automatically highlight the first error line found. To disable this feature, select *View...Editor Options* from the IDE main menu and uncheck automatically jump to first compilation error. At the time of writing, some compiler errors do not have line numbers bound to them. Under these circumstances, Proton IDE will be unable to automatically jump to the selected line.

Occasionally, the compiler will generate a valid Asm file but warnings or errors are generated during assembly. Proton IDE will display all assembler warnings or error messages in the error view, but you will be unable to automatically jump to a selected line.

Editor Options

The editor options dialog enables you to configure and control many of the Proton IDE features. The window is composed of four main areas, which are accessed by selecting the General, Highlighter, Program Header and Online Updating tabs.

Show Line Numbers in Left Gutter

Display line numbers in the editors left hand side gutter. If enabled, the gutter width is increased in size to accommodate a five digit line number.

Show Right Gutter

Displays a line to the right of the main editor. You can also set the distance from the left margin (in characters). This feature can be useful for aligning your program comments.

Use Smart Tabs

Normally, pressing the tab key will advance the cursor by a set number of characters. With smart tabs enabled, the cursor will move to a position along the current line which depends on the text on the previous line. Can be useful for aligning code blocks.

Convert Tabs to Spaces

When the tab key is pressed, the editor will normally insert a tab control character, whose size will depend on the value shown in the width edit box (the default is four spaces). If you then press the backspace key, the whole tab is deleted (that is, the cursor will move back four spaces). If convert tabs to spaces is enabled, the tab control character is replaced by the space control character (multiplied by the number shown in the width edit box). Pressing the backspace key will therefore only move the cursor back by one space. Please note that internally, the editor does not use hard tabs, even if convert tabs to spaces is unchecked.

Automatically Indent

When the carriage return key is pressed in the editor window, automatically indent will advance the cursor to a position just below the first word occurrence of the previous line. When this feature is unchecked, the cursor just moves to the beginning of the next line.

Show Parameter Hints

If this option is enabled, small prompts are displayed in the main editor window when a particular compiler keyword is recognised. For example,

DELAYMS

DELAYMS Value or Variable or Expression

Parameter hints are automatically hidden when the first parameter character is typed. To view the hint again, press F1. If you want to view more detailed context sensitive help, press F1 again.

Open Last File(s) When Application Starts

When checked, the documents that were open when Proton IDE was closed are automatically loaded again when the application is restarted.

Display Full Filename Path in Application Title Bar

By default, Proton IDE only displays the document filename in the main application title bar (that is, no path information is included). Check display full pathname if you would like to display additional path information in the main title bar.

Prompt if File Reload Needed

Proton IDE automatically checks to see if a file time stamp has changed. If it has (for example, and external program has modified the source code) then a dialog box is displayed asking if the file should be reloaded. If prompt on file reload is unchecked, the file is automatically reloaded without any prompting.

Automatically Select Variable on Code Explorer Click

By default, clicking on a link in the code explorer window will take you to the part of your program where a declaration has been made. Selecting this option will load the search name into the 'find dialog' search buffer. You then just need to press F3 to search for the next occurrence of the declaration in your program.

Automatically Jump to First Compilation Error

When this is enabled, Proton IDE will automatically jump to the first error line, assuming any errors are generated during compilation.

Automatically Change Identifiers to Match Declaration

When checked, this option will automatically change the identifier being typed to match that of the actual declaration. For example, if you have the following declaration,

Dim MyIndex as Byte

and you type 'myindex' in the editor window, Proton IDE will automatically change 'myindex' to 'MyIndex'. Identifiers are automatically changed to match the declaration even if the declaration is made in an include file.

Please note that the actual text is not physically changed, it just changes the way it is displayed in the editor window. For example, if you save the above example and load it into wordpad or another text editor, it will still show as 'myindex'. If you print the document, the identifier will be shown as 'MyIndex'. If you copy and paste into another document, the identifier will be shown as 'MyIndex', if the target application supports formatted text (for example Microsoft Word). In short, this feature is very useful for printing, copying and making you programs look consistent throughout.

Clear Undo History After Successful Compile

If checked, a successful compilation will clear the undo history buffer. A history buffer takes up system resources, especially if many documents are open at the same time. It's a good idea to have this feature enabled if you plan to work on many documents at the same time.

Display Full Summary After Successful Compile

If checked, a successful compilation will display a full summary in the results window. Disabling this option will still give a short summary in the IDE status bar, but the results window will not be displayed.

Default Source Folder

Proton IDE will automatically go to this folder when you invoke the file open or save as dialogs. To disable this feature, uncheck the 'Enabled' option, shown directly below the default source folder.

Highlighter Options

Item Properties

The syntax highlighter tab lets you change the colour and attributes (for example, bold and italic) of the following items: -

- Comment
- Device Name
- Identifier
- Keyword (Asm)
- Keyword (Declare)
- Keyword (Important)
- Keyword (Macro Parameter)
- Keyword (Proton)
- Keyword (User)
- Number
- Number (Binary)
- Number (Hex)
- SFR
- SFR (Bitname)
- String
- Symbol
- Preprocessor

The point size is ranged between 6pt to 16pt and is global. That is, you cannot have different point sizes for individual items.

Reserved Word Formatting

This option enables you to set how Proton IDE displays keywords. Options include: -

Database Default - the IDE will display the keyword as declared in the applications keyword database.

Uppercase - the IDE will display the keyword in uppercase.

Lowercase - the IDE will display the keyword in lowercase.

As Typed - the IDE will display the keyword as you have typed it.

Please note that the actual keyword text is not physically changed, it just changes the way it is displayed in the editor window. For example, if you save your document and load it into wordpad or another text editor, the keyword text will be displayed as you typed it. If you print the document, the keyword will be formatted. If you copy and paste into another document, the keyword will be formatted, if the target application supports formatted text (for example Microsoft Word).

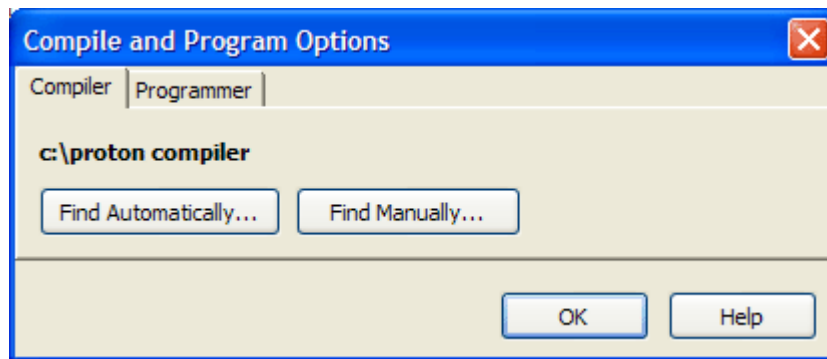
Header options allows you to change the author and copyright name that is placed in a header when a new document is created. For example: -

```
*****
' * Name      : Untitled.bas                      *
' * Author    : J.R Hartley                       *
' * Notice    : Copyright (c) 2011 MyCompany       *
' *           : All Rights Reserved               *
' * Date      : 06/03/11                          *
' * Version   : 1.0                               *
' * Notes     :                                    *
' *           :                                    *
*****
```

If you do not want to use this feature, simply deselect the enable check box.

Compile and Program Options

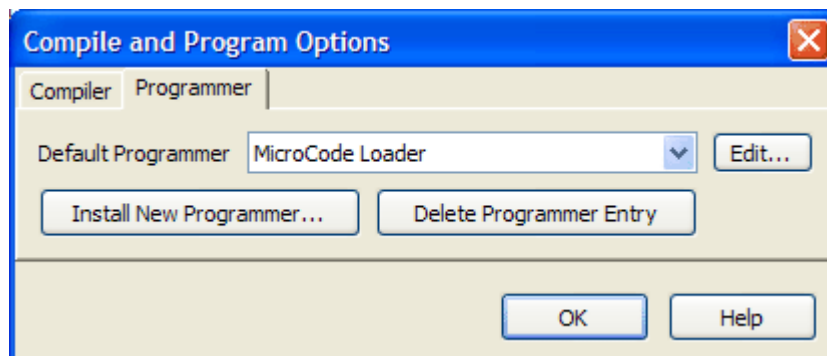
Compiler Tab



You can get the Proton IDE to locate a compiler directory automatically by clicking on the find automatically button. The auto-search feature will stop when a compiler is found.

Alternatively, you can select the directory manually by selecting the find manually button. The auto-search feature will search for a compiler and if one is found, the search is stopped and the path pointing to the compiler is updated. If you have multiple versions of a compiler installed on your system, use the find manually button. This ensures the correct compiler is used by the IDE.

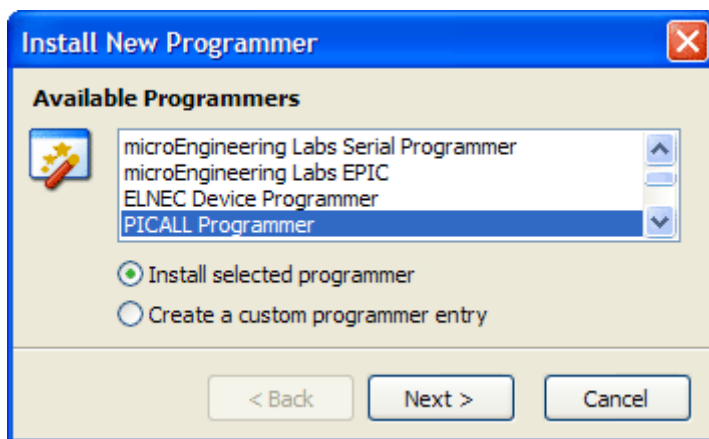
Programmer Tab



Use the programmer tab to install a new programmer, delete a programmer entry or edit the currently selected programmer. Pressing the Install New Programmer button will invoke the install new programmer wizard. The Edit button will invoke the install new programmer wizard in custom configuration mode.

Installing a Programmer

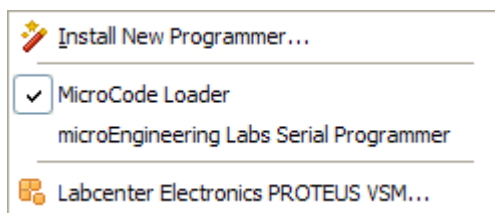
The Proton IDE enables you to start your preferred programming software from within the development environment. This enables you to compile and then program your microcontroller with just a few mouse clicks (or keyboard strokes, if you prefer). The first thing you need to do is tell Proton IDE which programmer you are using. Select View...Options from the main menu bar, then select the Programmer tab. Next, select the Add New Programmer button. This will open the install new programmer wizard.



Select the programmer you want Proton IDE to use, then choose the Next button. Proton IDE will now search your computer until it locates the required executable. If your programmer is not in the list, you will need to create a custom programmer entry.

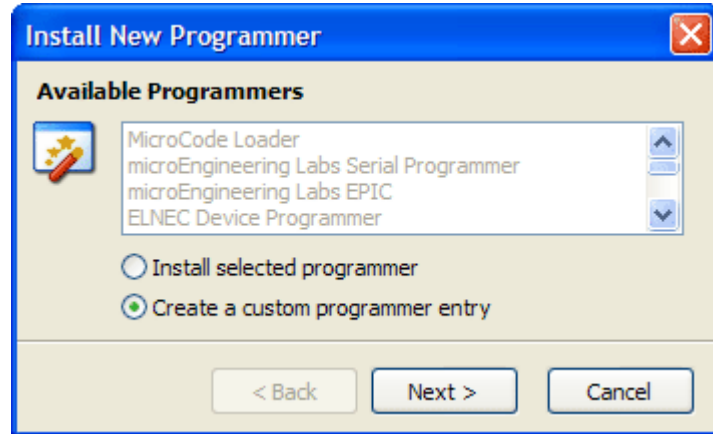
Your programmer is now ready for use. When you press the Compile and Program button on the main toolbar, your program is compiled and the programmer software started. The *.hex file name and target device is automatically set in the programming software (if this feature is supported), ready for you to program your microcontroller.

You can select a different programmer, or install another programmer, by pressing the small down arrow, located to the right of the compile and program button, as shown below



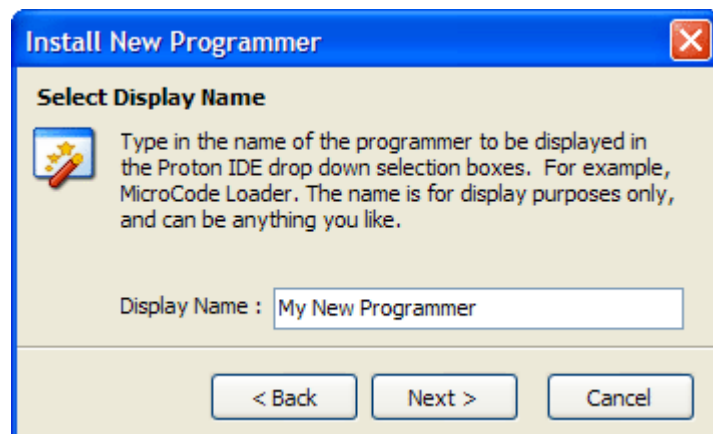
Creating a custom Programmer Entry

In most cases, Proton IDE has a set of pre-configured programmers available for use. However, if you use a programmer not included in this list, you will need to add a custom programmer entry. Select View...Options from the main menu bar, then select the Programmer tab. Next, select the Add New Programmer button. This will open the install new programmer wizard. You then need to select 'create a custom programmer entry', as shown below



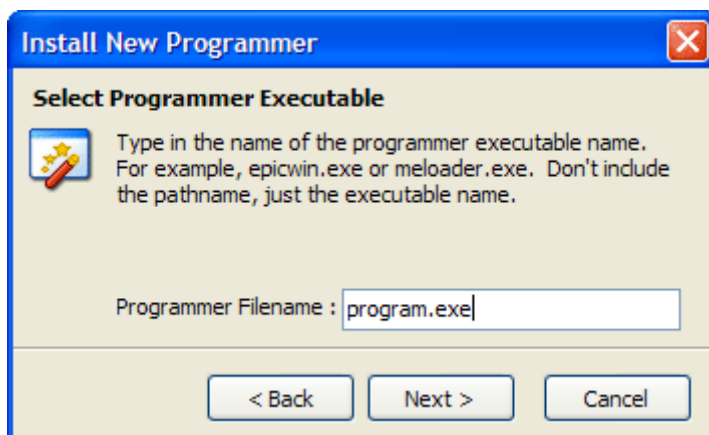
Select Display Name

The next screen asks you to enter the display name. This is the name that will be displayed in any programmer related drop down boxes. Proton IDE enables you to add and configure multiple programmers. You can easily switch from different types of programmer from the compile and program button, located on the main editor toolbar. The multiple programmer feature means you do not have to keep reconfiguring your system when you switch programmers. Proton IDE will remember the settings for you. In the example below, the display name will be 'My New Programmer'.



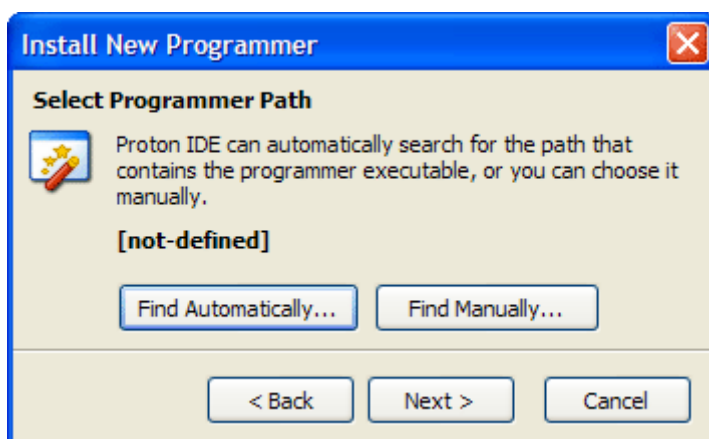
Select Programmer Executable

The next screen asks for the programmer executable name. You do not have to give the full path, just the name of the executable name will do.



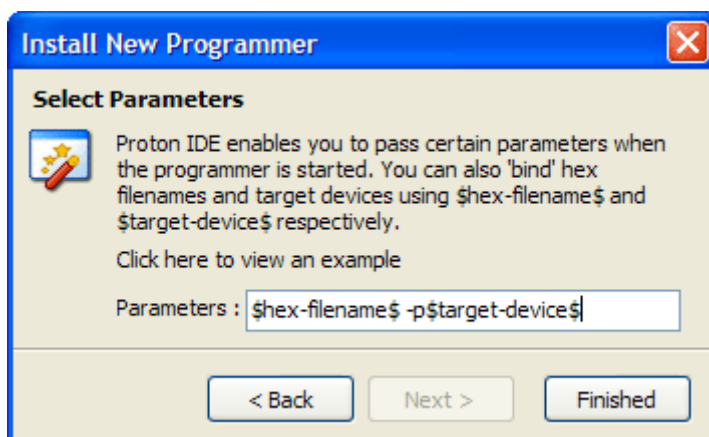
Select Programmer Path

The next screen is the path to the programmer executable. You can let Proton IDE find it automatically, or you can select it manually.



Select Parameters

The final screen is used to set the parameters that will be passed to your programmer. Some programmers, for example, EPICWin™ allows you to pass the device name and hex filename. Proton IDE enables you to 'bind' the currently selected device and *.hex file you are working on.



For example, if you are compiling 'blink.bas' in the Proton IDE using a 16F628, you would want to pass the 'blink.hex' file to the programmer and also the name of the microcontroller you intend to program. Here is the EPICWin™ example: -

```
-pPIC$target-device$ $hex-filename$
```

When EPICWin™ is started, the device name and hex filename are 'bound' to \$target-device\$ and \$hex-filename\$ respectively. In the 'blink.bas' example, the actual parameter passed to the programmer would be: -

```
-pPIC16F628 blink.hex
```

Parameter Summary

Parameter	Description
\$target-device\$	Microcontroller name
\$hex-filename\$	Hex filename and path, DOS 8.3 format
\$long-hex-filename\$	Hex filename and path
\$asm-filename\$	Asm filename and path, DOS 8.3 format
\$long-asm-filename\$	Asm filename and path

Microcode Loader

The PIC16F87x(A), 16F8x and PIC18Fxxx(x) series of microcontrollers have the ability to write to their own program memory, without the need of a hardware programmer. A small piece of software called a bootloader resides on the target microcontroller, which allows user code and eeprom data to be transmitted over a serial cable and written to the device. The MicroCode Loader application is the software which resides on the computer. Together, these two components enable a user to program, verify and read their program and eeprom data all in circuit.

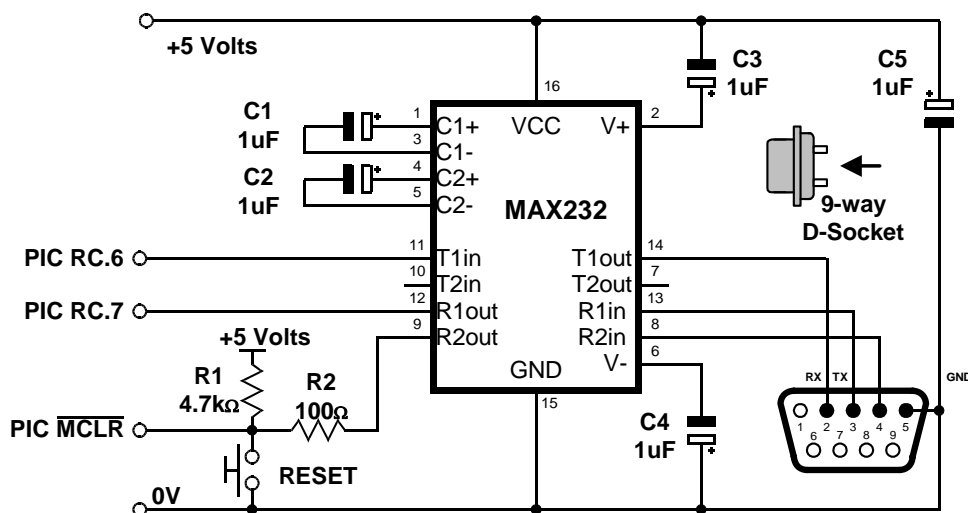
When power is first applied to the microcontroller (or it is reset), the bootloader first checks to see if the MicroCode Loader application has something for it to do (for example, program your code into the target device). If it does, the bootloader gives control to MicroCode Loader until it is told to exit. However, if the bootloader does not receive any instructions with the first few hundred milliseconds of starting, the bootloader will exit and the code previously written to the target device will start to execute.

The bootloader software resides in the upper 256 words of program memory (336 words for 18Fxxx devices), with the rest of the microcontroller code space being available for your program. All eeprom data memory and microcontroller registers are available for use by your program. Please note that only the program code space and eeprom data space may be programmed, verified and read by MicroCode Loader. The microcontroller ID location and configuration fuses are not available to the loader process. Configuration fuses must therefore be set at the time the bootloader software is programmed into the target microcontroller.

Hardware Requirements

MicroCode Loader communicates with the target microcontroller using its hardware Universal Synchronous Asynchronous Receiver Transmitter (USART). You will therefore need a development board that supports RS232 serial communication in order to use the loader. There are many boards available which support RS232.

Whatever board you have, if the board has a 9 pin serial connector on it, the chances are it will have a MAX232 or equivalent located on the board. This is ideal for MicroCode Loader to communicate with the target device using a serial cable connected to your computer. Alternatively, you can use the following circuit and build your own.



Note: Components R1, R2, and the Reset switch are optional, and serve to reset the microcontroller automatically. If these components are not used, the connections to R2in and R2out of the MAX232 may be omitted.

MicroCode Loader supports a host of devices capable of using a bootloader and the support will grow as new devices become available.

MicroCode Loader comes with a number of pre-compiled *.hex files, ready for programming into the target microcontroller. If you require a bootloader file with a different configuration, please contact Mecanique.

Using the Bootloader is very easy. Before using this guide make sure that your target microcontroller is supported by the loader and that you also have suitable hardware.

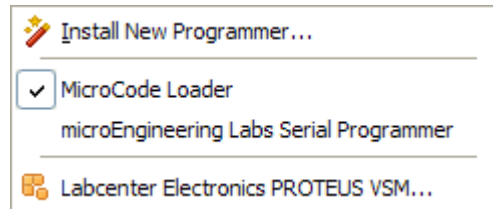
Programming the Loader Firmware

Before using the Bootloader, you need to ensure that the bootloader firmware has been programmed onto the target microcontroller using a hardware programmer. This is a one off operation, after which you can start programming your target device over an RS232 serial connection. You need to make sure that the bootloader *.hex file matches the clock speed of your target microcontroller. For example, if you are using a 16F877 on a development board running at 20MHz, then you need to use the firmware file called 16F877_20.hex. If you don't do this, the Bootloader will be unable to communicate with the target microcontroller. The Compiler comes with a number of pre-compiled *.hex files, ready for programming into the target microcontroller. The loader firmware files can be found in the MCLoader folder, located in your main IDE installation folder. Default fuse settings are embedded in the firmware *.hex file. You should not normally change these default settings. You should certainly never select the code protect fuse. If the code protect fuse is set the Bootloader will be unable to program your *.hex file.

Configuring the Loader

Assuming you now have the firmware installed on your microcontroller, you now just need to tell MicroCode Loader which COM port you are going to use. To do this, select View...Loader from the MicroCode IDE main menu. Select the COM port from the MicroCode Loader main toolbar. Finally, make sure that MicroCode Loader is set as your default programmer.

Click on the down arrow, to the right of the Compile and Program button. Check the MicroCode Loader option, like this: -



Using MicroCode Loader

Connect a serial cable between your computer and development board. Apply power to the board.

Press 'Compile and Program' or F10 to compile your program. If there are no compilation errors, the MicroCode Loader application will start. It may ask you to reset the development board in order to establish communications with the resident microcontroller bootloader. This is perfectly normal for development boards that do not implement a software reset circuit. If required, press reset to establish communications and program your microcontroller.

Loader Options

Loader options can be set by selecting the Options menu item, located on the main menu bar.

Program Code

Optionally program user code when writing to the target microcontroller. Uncheck this option to prevent user code from being programmed. The default is On.

Program Data

Optionally program Eeprom data when writing to the target microcontroller. Uncheck this option to prevent Eeprom data from being programmed. The default is On.

Verify Code When Programming

Optionally verify a code write operation when programming. Uncheck this option to prevent user code from being verified when programming. The default is On.

Verify Data When Programming

Optionally verify a data write operation when programming. Uncheck this option to prevent user data from being verified when programming. The default is On.

Verify Code

Optionally verify user code when verifying the loaded *.hex file. Uncheck this option to prevent user code from being verified. The default is On.

Verify Data

Optionally verify Eeprom data when verifying the loaded *.hex file. Uncheck this option to prevent Eeprom data from being verified. The default is On.

Verify After Programming

Performs an additional verification operation immediately after the target microcontroller has been programmed. The default is Off.

Run User Code After Programming

Exit the bootloader process immediately after programming and then start running the target user code. The default is On.

Load File Before Programming

Optionally load the latest version of the *.hex file immediately before programming the target microcontroller. The default is Off.

Baud Rate

Select the speed at which the computer communicates with the target microcontroller. By default, the Auto Detect option is enabled. This feature enables MicroCode Loader to determine the speed of the target microcontroller and set the best communication speed for that device.

If you select one of the baud rates manually, it must match the baud rate of the loader software programmed onto the target microcontroller. For devices running at less than 20MHz, this is 19200 baud. For devices running at 20MHz, you can select either 19200 or 115200 baud.

Loader Main Toolbar



Open Hex File

The open button loads a *.hex file ready for programming.



Program

The program button will program the loaded hex file code and eeprom data into the target microcontroller. When programming the target device, a verification is normally done to ensure the integrity of the programmed user code and eeprom data. You can override this feature by un-checking either Verify Code When Programming or Verify Data When Programming. You can also optionally verify the complete *.hex file after programming by selecting the Verify After Programming option.

Pressing the program button will normally program the currently loaded *.hex file. However, you can load the latest version of the *.hex file immediately before programming by checking Load File Before Programming option. You can also set the loader to start running the user code immediately after programming by checking the Run User Code After Programming option. When programming the target device, both user code and eeprom data are programmed by default (recommended). However, you may want to just program code or eeprom data. To change the default configuration, use the Program Code and Program Data options.

Should any problems arise when programming the target device, a dialog window will be displayed giving additional details. If no problems are encountered when programming the device, the status window will close at the end of the write sequence.



Read

The read button will read the current code and eeprom data from the target microcontroller. Should any problems arise when reading the target device, a dialog window will be displayed giving additional details. If no problems are encountered when reading the device, the status window will close at the end of the read sequence.



Verify

The verify button will compare the currently loaded *.hex file code and eeprom data with the code and eeprom data located on the target microcontroller. When verifying the target device, both user code and eeprom data are verified by default. However, you may want to just verify code or eeprom data. To change the default configuration, use the Verify Code and Verify Data options.

Should any problems arise when verifying the target device, a dialog window will be displayed giving additional details. If no problems are encountered when verifying the device, the status window will close at the end of the verification sequence.



Erase

The erase button will erase all of the code memory on a PIC16F8x and PIC18Fxxx(x) microcontroller.



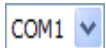
Run User Code

The run user code button will cause the bootloader process to exit and then start running the program loaded on the target microcontroller.



Loader Information

The loader information button displays the loader firmware version and the name of the target microcontroller, for example PIC16F877.



Loader Serial Port

The loader serial port drop down box allows you to select the com port used to communicate with the target microcontroller.

IDE Plugins

The Proton IDE has been designed with flexibility in mind. Plugins enable the functionality of the IDE to be extended by through additional third party software, which can be integrated into the development environment. Proton IDE comes with a default set of plugins which you can use straight away. These are: -

- ASCII Table
- Assembler
- Hex View
- Serial Communicator
- Labcenter Electronics Proteus VSM

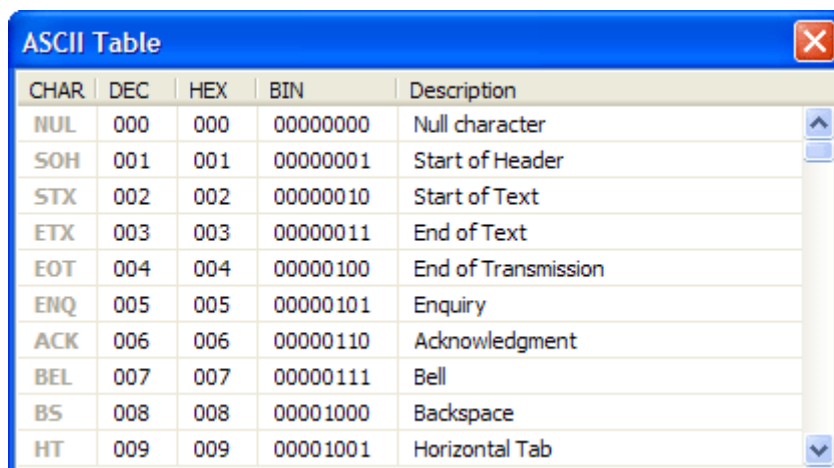
To access a plugin, select the plugin icon just above the main editor window. A drop down list of available plugins will then be displayed. Plugins can also be selected from the main menu, or by right clicking on the main editor window.

Plugin Developer Notes

The plugin architecture has been designed to make writing third party plugins very easy, using the development environment of your choice (for example Visual BASIC, C++ or Borland Delphi). This architecture is currently evolving and is therefore publicly undocumented until all of the protocols have been finalised. As soon as the protocol details have been finalised, this documentation will be made public. For more information, please feel free to contact us.

ASCII Table

The American Standard Code for Information Interchange (ASCII) is a set of numerical codes, with each code representing a single character, for example, 'a' or '\$'.

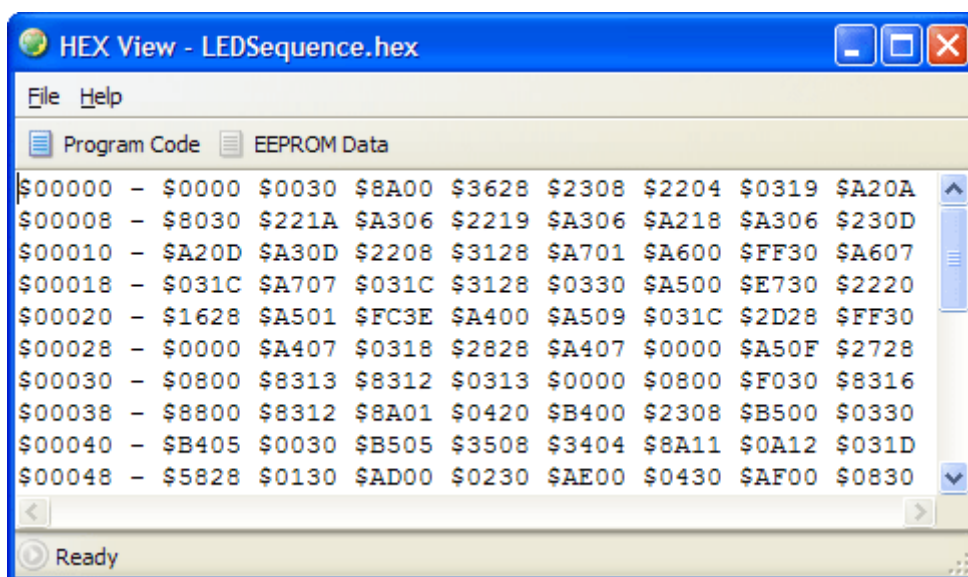


CHAR	DEC	HEX	BIN	Description
NUL	000	000	00000000	Null character
SOH	001	001	00000001	Start of Header
STX	002	002	00000010	Start of Text
ETX	003	003	00000011	End of Text
EOT	004	004	00000100	End of Transmission
ENQ	005	005	00000101	Enquiry
ACK	006	006	00000110	Acknowledgment
BEL	007	007	00000111	Bell
BS	008	008	00001000	Backspace
HT	009	009	00001001	Horizontal Tab

The ASCII table plugin enables you to view these codes in either decimal, hexadecimal or binary. The first 32 codes (0..31) are often referred to as non-printing characters, and are displayed as grey text.

Hex View

The Hex view plugin enables you to view program code and EEPROM data for 14 and 16 core devices.



The Hex View window is automatically updated after a successful compile, or if you switch program tabs in the IDE. By default, the Hex view window remains on top of the main IDE window. To disable this feature, right click on the Hex View window and uncheck the Stay on Top option.

Assembler Window

The Assembler plugin allows you to view and modify the *.asm file generated by the compiler. Using the Assembler window to modify the generated *.asm file is not really recommended, unless you have some experience using assembler.

Assembler Menu Bar

File Menu

New - Creates a new document. A header is automatically generated, showing information such as author, copyright and date.

- **Open** - Displays a open dialog box, enabling you to load a document into the Assembler plugin. If the document is already open, then the document is made the active editor page.
- **Save** - Saves a document to disk. This button is normally disabled unless the document has been changed. If the document is 'untitled', a save as dialog is invoked. A save as dialog is also invoked if the document you are trying to save is marked as read only.
- **Save As** - Displays a save as dialog, enabling you to name and save a document to disk.
- **Close** - Closes the currently active document.
- **Close All** - Closes all editor documents and then creates a new editor document.
- **Reopen** - Displays a list of Most Recently Used (MRU) documents.
- **Print Setup** - Displays a print setup dialog.
- **Print** - Prints the currently active editor page.
- **Exit** - Enables you to exit the Assembler plugin.

Edit Menu

- **Undo** - Cancels any changes made to the currently active document page.
- **Redo** - Reverse an undo command.
- **Cut** - Cuts any selected text from the active document page and places it into the clipboard.
- **Copy** - Copies any selected text from the active document page and places it into the clipboard.
- **Paste** - Paste the contents of the clipboard into the active document page. This option is disabled if the clipboard does not contain any suitable text.
- **Delete** - Deletes any selected text. This option is disabled if no text has been selected.
- **Select All** - Selects the entire text in the active document page.

- **Find** - Displays a find dialog.
- **Replace** - Displays a find and replace dialog.
- **Find Next** - Automatically searches for the next occurrence of a word. If no search word has been selected, then the word at the current cursor position is used. You can also select a whole phrase to be used as a search term. If the editor is still unable to identify a search word, a find dialog is displayed.

View Menu

- **Options** - Displays the application editor options dialog.
- **Toolbars** - Display or hide the main and assemble and program toolbars. You can also toggle the toolbar icon size.

Help Menu

- **Help Topics** - Displays the IDE help file.
- **About** - Display about dialog, giving the Assembler plugin version number.

Assembler Main Toolbar



New

Creates a new document. A header is automatically generated, showing information such as author, copyright and date.



Open

Displays a open dialog box, enabling you to load a document into the Assembler plugin. If the document is already open, then the document is made the active editor page.



Save

Saves a document to disk. This button is normally disabled unless the document has been changed. If the document is 'untitled', a save as dialog is invoked. A save as dialog is also invoked if the document you are trying to save is marked as read only.



Cut

Cuts any selected text from the active document page and places it into the clipboard. This option is disabled if no text has been selected.



Copy

Copies any selected text from the active document page and places it into the clipboard. This option is disabled if no text has been selected.



Paste

Paste the contents of the clipboard into the active document page. This option is disabled if the clipboard does not contain any suitable text.



Undo

Cancels any changes made to the currently active document page.



Reverse an undo command.

Assembler Editor Options

Show Line Numbers in Left Gutter

Display line numbers in the editors left hand side gutter. If enabled, the gutter width is increased in size to accommodate a five digit line number.

Show Right Gutter

Displays a line to the right of the main editor. You can also set the distance from the left margin (in characters). This feature can be useful for aligning your program comments.

Use Smart Tabs

Normally, pressing the tab key will advance the cursor by a set number of characters. With smart tabs enabled, the cursor will move to a position along the current line which depends on the text on the previous line. Can be useful for aligning code blocks.

Convert Tabs to Spaces

When the tab key is pressed, the editor will normally insert a tab control character, whose size will depend on the value shown in the width edit box (the default is four spaces). If you then press the backspace key, the whole tab is deleted (that is, the cursor will move back four spaces). If convert tabs to spaces is enabled, the tab control character is replaced by the space control character (multiplied by the number shown in the width edit box). Pressing the backspace key will therefore only move the cursor back by one space. Please note that internally, the editor does not use hard tabs, even if convert tabs to spaces is unchecked.

Automatically Indent

When the carriage return key is pressed in the editor window, automatically indent will advance the cursor to a position just below the first word occurrence of the previous line. When this feature is unchecked, the cursor just moves to the beginning of the next line.

Show Parameter Hints

If this option is enabled, small prompts are displayed in the main editor window when a particular compiler keyword is recognised.

Open Last File(s) When Application Starts

When checked, the documents that were open when the Assembler plugin was closed are automatically loaded again when the application is restarted.

Display Full Filename Path in Application Title Bar

By default, the Assembler plugin only displays the document filename in the main application title bar (that is, no path information is included). Check display full pathname if you would like to display additional path information in the main title bar.

Prompt if File Reload Needed

The Assembler plugin automatically checks to see if a file time stamp has changed. If it has (for example, and external program has modified the source code) then a dialog box is displayed asking if the file should be reloaded. If prompt on file reload is unchecked, the file is automatically reloaded without any prompting.

Automatically Jump to First Compilation Error

When this is enabled, the Assembler plugin will automatically jump to the first error line, assuming any errors are generated during compilation.

Clear Undo History After Successful Compile

If checked, a successful compilation will clear the undo history buffer. A history buffer takes up system resources, especially if many documents are open at the same time. It's a good idea to have this feature enabled if you plan to work on many documents at the same time.

Default Source Folder

The Assembler plugin will automatically go to this folder when you invoke the file open or save as dialogs. To disable this feature, uncheck the 'Enabled' option, shown directly below the default source folder.

Serial Communicator

The Serial Communicator plugin is a simple to use utility which enables you to transmit and receive data via a serial cable connected to your PC and development board. The easy to use configuration window allows you to select port number, baudrate, parity, byte size and number of stop bits. Alternatively, you can use Serial Communicator favourites to quickly load pre-configured connection settings.

Menu options

File Menu

- **Clear** - Clears the contents of either the transmit or receive window.
- **Open** - Displays a open dialog box, enabling you to load data into the transmit window.
- **Save As** - Displays a save as dialog, enabling you to name and save the contents of the receive window.
- **Exit** - Enables you to exit the Serial Communicator software.

Edit Menu

- **Undo** - Cancels any changes made to either the transmit or receive window.
- **Cut** - Cuts any selected text from either the transmit or receive window.

- **Copy** - Copies any selected text from either the transmit or receive window.
- **Paste** - Paste the contents of the clipboard into either the transmit or receive window. This option is disabled if the clipboard does not contain any suitable text.
- **Delete** - Deletes any selected text. This option is disabled if no text has been selected.

View Menu

- **Configuration Window** - Display or hide the configuration window.
- **Toolbars** - Display small or large toolbar icons.

Help Menu

- **Help Topics** - Displays the serial communicator help file.
- **About** - Display about dialog, giving software version information.

Serial Communicator Main Toolbar



Clear

Clears the contents of either the transmit or receive window.



Open

Displays a open dialog box, enabling you to load data into the transmit window.



Save As

Displays a save as dialog, enabling you to name and save the contents of the receive window.



Cut

Cuts any selected text from either the transmit or receive window.



Copy

Copies any selected text from either the transmit or receive window.



Paste

Paste the contents of the clipboard into either the transmit or receive window. This option is disabled if the clipboard does not contain any suitable text.



Connect

Connects the Serial Communicator software to an available serial port. Before connecting, you should ensure that your communication options have been configured correctly using the configuration window.

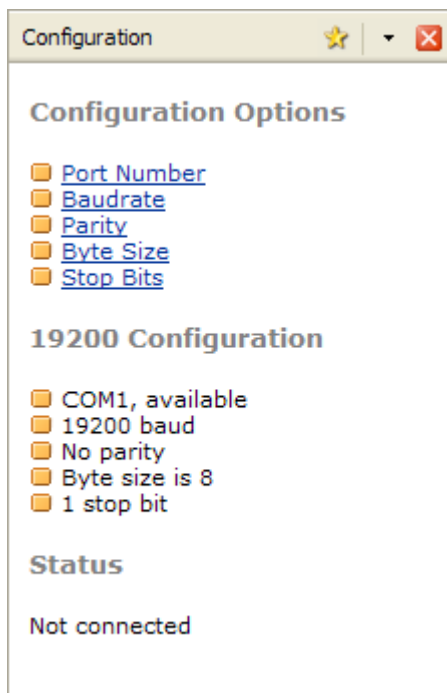


Disconnect

Disconnect the Serial Communicator from a serial port.

Configuration

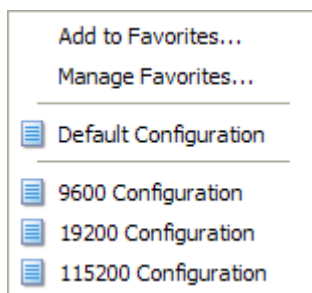
The configuration window is used to select the COM port you want to connect to and also set the correct communications protocols.



Clicking on a configuration link will display a drop down menu, listing available options. A summary of selected options is shown below the configuration links. For example, in the image above, summary information is displayed under the heading 19200 Configuration.

★ Favourites

Pressing the favourite icon will display a number of options allowing you to add, manage or load configuration favourites.



Add to Favourites

Select this option if you wish to save your current configuration. You can give your configuration a unique name, which will be displayed in the favourite drop down menu. For example, 9600 Configuration or 115200 Configuration

Manage Favourites

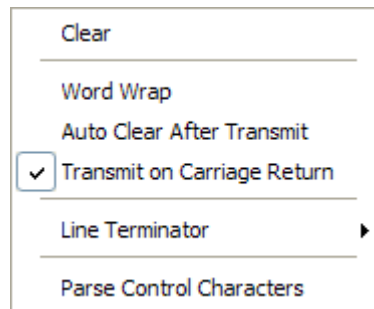
Select this option to remove a previously saved configuration favourite.

Notes

After pressing the connect icon on the main toolbar, the configuration window is automatically closed and opened again when disconnect is pressed. If you don't want the configuration window to automatically close, right click on the configuration window and un-check the Auto-Hide option.

Transmit Window

The transmit window enables you to send serial data to an external device connected to a PC serial port. In addition to textual data, the send window also enables you to send control characters. To display a list of transmit options, right click on the transmit window.



Clear

Clear the contents of the transmit window.

Word Wrap

This option allows you to wrap the text displayed in the transmit window.

Auto Clear After Transmit

Enabling this option will automatically clear the contents of the transmit window when data is sent.

Transmit on Carriage Return

This option will automatically transmit data when the carriage return key is pressed. If this option is disabled, you will need to manually press the send button or press F4 to transmit.

Line Terminator

You can append your data with a number of line terminations characters. These include CR, CR and LF, LF and CR, null and No Terminator.

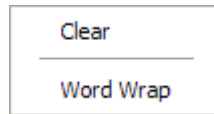
Parse Control Characters

When enabled, the parse control characters option enables you to send control characters in your message, using either a decimal or hexadecimal notation. For example, if you want to send hello world followed by a carriage return and line feed character, you would use hello world#13#10 for decimal, or hello world\$D\$A for hex. Only numbers in the range 0 to 255 will be converted. For example, sending the message letter #9712345 will be interpreted as letter a12345.

If the sequence of characters does not form a legal number, the sequence is interpreted as normal characters. For example, hello world#here I am. If you don't want characters to be interpreted as a control sequence, but rather send it as normal characters, then all you need to do is use the tilde symbol (~). For example, letter ~#9712345 would be sent as letter #9712345.

Receive Window

The receive window is used to capture data sent from an external device (for example, a PIC MCU) to your PC. To display a list of transmit options, right click on the receive window.



Clear

Clear the contents of the receive window.

Word Wrap

When enabled, incoming data is automatically word wrapped.

Notes

In order to advance the cursor to the next line in the receive window, you must transmit either a CR (\$D) or a CR LF pair (\$D \$A) from your external device.

Labcenter Electronics Proteus VSM

Proteus Virtual System Modelling (VSM) combines mixed mode SPICE circuit simulation, animated components and microprocessor models to facilitate co-simulation of complete micro-controller based designs. For the first time ever, it is possible to develop and test such designs before a physical prototype is constructed.

The Proton Plus Development Suite comes shipped with a free demonstration version of the Proteus simulation environment and also a number of pre-configured Virtual Hardware Boards (VHB). Unlike the professional version of Proteus, you are unable to make any changes to the pre-configured boards or create your own boards.

If you already have a full version of Proteus VSM installed on your system (6.5.0.5 or higher), then this is the version that will be used by the IDE. If you don't have the full version, the IDE will default to using the demonstration installation.

System Requirements

Windows XP or Vista

512MB RAM (1 GB or higher recommended)

500 MHz Processor

Further Information

You can find out more about the simulator supplied with the Proton Development Suite from Labcenter Electronics

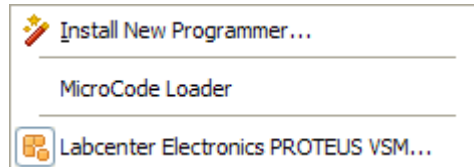
ISIS Simulator Quick Start Guide

This brief tutorial aims to outline the steps you need to take in order to use Labcenter Electronics Proteus Virtual System Modelling (VSM) with the Proton IDE. The first thing you need to do is load or create a program to simulate. In this worked example, we will keep things simple and use a classic flashing LED program. In the IDE, press the New toolbar button and type in the following: -

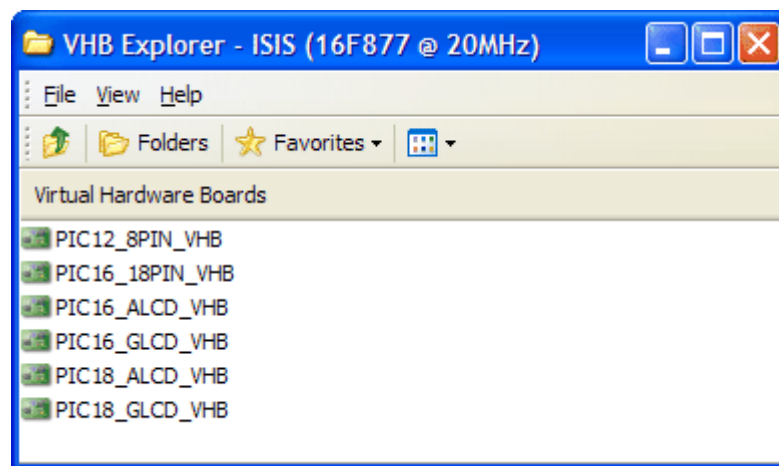
```
Device = 16F877
Declare Xtal = 20
Symbol LED = PORTD.0
MainProgram:
High LED
```

```
Delaysms 500
Low LED
Delaysms 500
Goto MainProgram
```

You now need to make sure that the output of the compile and program process is re-directed to the simulator. Normally, pressing compile and program will create a *.hex file which is then sent to your chosen programmer. However, we want the output to be sent to the simulator, not a device programmer. To do this, press the small down arrow to the right of the compile and program toolbar icon and check the Labcenter Electronics Proteus VSM option, as shown below: -



After selecting the above option, save your program and then press the compile and program toolbar button to build your project. This will then start the Virtual Hardware Board (VHB) Explorer, as shown below: -



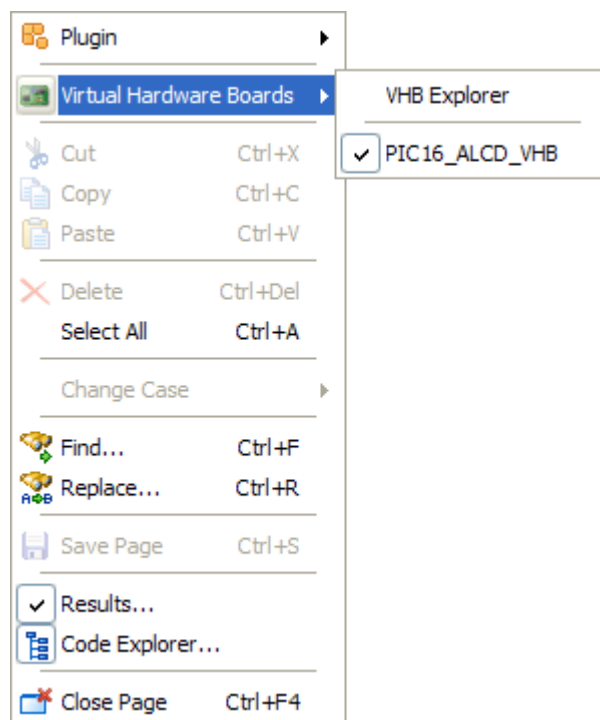
VHB Explorer is the IDE plugin that co-ordinates activity between the IDE and the simulator. Its primary purpose is to bind a Virtual Hardware Board to your program. In this example, the program has been built for the 16F877 MCU which flashes an LED connected to PORTD.0. To run the simulation for this program, just double click on the PIC16_ALCD_VHB hardware board item. This will invoke the Proteus simulator which will then automatically start executing your program using the selected board.

Additional Integration Tips


If you followed the Proteus VSM quick start guide, you will know how easy it is to load your program into the simulation environment with the Virtual Hardware Board (VHB) Explorer. However, one thing you might have noticed is that each time you press compile and program the VHB Explorer is always displayed. If you are using the same simulation board over and over again, manually having to select the board using VHB Explorer can become a little tiresome.

Virtual Hardware Boards Favourites

The good news is that every time you select a board using VHB Explorer, it is saved as a VHB Explorer favourite. You can access VHB Explorer favourites from within Proton IDE by right clicking on the main editor window and selecting the Virtual Hardware Boards option, as shown below :-



In the quick start guide, the program was bound to a simulation board called PIC16_ALCD_VHB. If we check this favourite and then press compile and program, VHB Explorer is not displayed. Instead, you project is loaded immediately into the Proteus simulation environment. You can have more than one board bound to your project, allowing you to quickly switch between target simulation boards during project development.

To add additional boards to your project, manually start VHB Explorer by selecting the plugin icon  and clicking on the Labcenter Electronics Proteus VSM... option. When VHB Explorer starts, just double click on the board you want to be bound to your current project. Your new board selection will be displayed next time you right click on the main editor window and select Virtual Hardware Boards. You can delete a favourite board by manually starting VHB Explorer and pressing the Favourites toolbar icon. Choose the Manage Favourites option to remove the virtual hardware board from the favourites list.

Compiler Overview.

PICmicro Devices

The compiler supports most of the PICmicro™ range of devices, and takes full advantage of their various features e.g. A/D converter, data memory eeprom area, hardware multiply etc.

This manual is not intended to give details about individual microcontroller devices, therefore, for further information visit the Microchip website at www.microchip.com, and download the multitude of datasheets and application notes available.

Limited 12-bit Device Compatibility.

The 12-bit core microcontrollers have been available for a long time, and are at the heart of many excellent, and complex projects. However, with their limited architecture, they were never intended to be used for high level languages such as BASIC. Some of these limits include only a two-level hardware stack and small amounts of general purpose RAM memory. The code page size is also small at 512 bytes. There is also a limitation that calls and computed jumps can only be made to the first half (256 words) of any code page. Therefore, these limitations have made it necessary to eliminate some compiler commands and modify the operation of others.

While many useful programs can be written for the 12-bit core devices using the compiler, there will be some applications that are not suited to them. Choosing a 14-bit core device with more resources will, in most instances, be the best solution, or better still, choose a suitable 18F device.

Some of the commands that are not supported™ for the 12-bit core devices are illustrated in the table below: -

Command	Reason for omission
Dwords	Memory limitations
Floats	Memory limitations
Signed Variables	Memory limitations
Adin	No internal ADCs
Cdata	No write modify feature
Cls	Limited stack size
Cread	No write modify feature
Cursor	Limited stack size
Cwrite	No write modify feature
DTMFout	Limited stack size
Edata	No on-board EEPROM
Eread	No on-board EEPROM
Ewrite	No on-board EEPROM
Freqout	Limited stack size
LCDread	No graphic LCD support
LCDwrite	No graphic LCD support
Hpwm	No 12-bit MSSP modules
Hrsin	No hardware serial port
Hrsout	No hardware serial port
Hserin	No hardware serial port
Hserout	No hardware serial port
Interrupts	No Interrupts
Pixel	No graphic LCD support
Plot	No graphic LCD support

Serout	Limited memory
Serin	Limited memory
Sound2	Limited resources
UnPlot	No graphic LCD support
USBin	No 12-bit USB devices
USBout	No 12-bit USB devices
Xin	Limited stack size
Xout	Limited stack size

Trying to use any of the above commands with 12-bit core devices will result in the compiler producing numerous Syntax errors. If any of these commands are a necessity, then choose a comparable standard or enhanced 14-bit core device.

The available commands that have had their operation modified are: -

Print, Rsout, Busin, Busout

Most of the modifiers are not supported for these commands because of memory and stack size limitations, this includes the **At**, and the **Str** modifier. However, the **@**, **Dec** and **Dec3** modifiers are still available.

Programming Considerations for 12-bit core Devices.

Because of the limited architecture of the 12-bit core microcontrollers, programs compiled for them by the compiler will be larger and slower than programs compiled for the 14-bit core devices. The two main programming limitations that will most likely occur are running out of RAM memory for variables, and running past the first 256 word limit for the library routines.

Even though the compiler arranges its internal system variables more intuitively than previous versions, it still needs to create temporary variables for complex expressions etc. It also needs to allocate extra RAM for use as a Software-Stack so that the BASIC program is still able to nest **Gosubs** up to 4 levels deep.

Some devices only have 25 bytes of RAM so there is very little space for user variables on those devices. Therefore, use variables sparingly, and always use the appropriately sized variable for a specific task. i.e. **Byte** variable if 0-255 is required, **Word** variable if 0-65535 required, **Bit** variables if a true or false situation is required. Try to alias any commonly used variables, such as loops or temporary stores etc.

As was mentioned earlier, 12-bit core microcontrollers can call only into the first half (256 words) of a code page. Since the compiler's library routines are all accessed by calls, they must reside entirely in the first 256 words of the code space. Many library routines, such as **Busin**, are quite large. It may only take a few routines to outgrow the first 256 words of code space. There is no work around for this, and if it is necessary to use more library routines that will fit into the first half of the first code page, it will be necessary to move to a 14-bit core device instead of the 12-bit core device.

No 32-bit or floating point variable support with 12-bit core devices.

Because of the profound lack of RAM space available on most 12-bit core devices, the Proton compiler does not allow 32-bit **Dword** type variables to be used. For 32-bit support, use one of the many 14-bit core, or 18F equivalent devices. Floating point variables are also not supported with 12-bit core devices.

Device Specific issues

Before venturing into your latest project, always read the datasheet for the specific device being used. Because some devices have features that may interfere with expected pin operations. The PIC16C62x and the 16F62x devices are examples of this. These devices have analogue comparators on PORTA. When these chips first power up, PORTA is set to analogue mode. This makes the pin functions on PORTA work in a strange manner. To change the pins to digital, simply add the following line near the front of your BASIC program, or before any of the pins are accessed: -

```
CMCON = 7
```

Any device with analogue inputs will power up in analogue mode. If you intend to use them as digital types you must set the pins to digital by using the following declare: -

```
Declare All_Digital = True
```

This will set analogue pins to digital on any compatible device.

Alternatively, SFRs (Special Function Registers) can be manipulated: -

```
ADCON1 = 7
```

Note that not all devices require the same registers manipulated and the datasheet should always be consulted before attempting this for the first time.

Another example of potential problems is that bit-4 of PORTA (PORTA.4) exhibits unusual behaviour when used as an output. This is because the pin has an open drain output rather than the usual bipolar stage as in the rest of the output pins. This means it can pull to ground when set to 0 (low), but it will simply float when set to a 1 (high), instead of going high.

To make this pin act as expected, add a pull-up resistor between the pin and 5 Volts. A typical value resistor may be between 1K Ω and 33K Ω , depending on the device it is driving. If the pin is used as an input, it behaves the same as any other pin.

Some devices, such as the PIC16F87x range, allow low-voltage programming. This function takes over one of the PORTB (PORTB.3) pins and can cause the device to act erratically if this pin is not pulled low. In normal use, It's best to make sure that low-voltage programming is disabled at the time the device is programmed. By default, the low voltage programming fuse is disabled, however, if the **Config** directive is used, then it may inadvertently be omitted.

All of the microcontroller's pins are set to inputs on power-up. If you need a pin to be an output, set it to an output before you use it, or use a BASIC command that does it for you. Once again, always read the PICmicro™ data sheets to become familiar with the particular part.

The name of the port pins on the 8 pin devices such as the PIC12C50X, PIC12C67x ,12CE67x and 12F675 is GPIO. The name for the Tris register is TrisIO: -

```
GPIO.0 = 1           ' Set GPIO.0 high
TRISIO = %101010    ' Manipulate ins and outs
```

However, these are also mapped as PORTB, therefore any reference to PORTB on these devices will point to the relevant pin.

Identifiers

An identifier is a technical term for a name. Identifiers are used for line labels, variable names, and constant aliases. An identifier is any sequence of letters, digits, and underscores, although it must not start with a digit. Identifiers are not case sensitive, therefore label, LABEL, and Label are all treated as equivalent. And while labels might be any number of characters in length, only the first 32 are recognised.

Line Labels

In order to mark statements that the program may wish to reference with the **Goto**, **Call**, or **Go-sub** commands, the compiler uses line labels. Unlike many older BASICs, the compiler does not allow or require line numbers and doesn't require that each line be labelled. Instead, any line may start with a line label, which is simply an identifier followed by a colon ':'.

```
Label:  
  Print "Hello World"  
  Goto Label
```


Variables

Variables are where temporary data is stored in a BASIC program. They are created using the **Dim** keyword. Because RAM space on 8-bit microcontrollers is somewhat limited, choosing the right size variable for a specific task is important. Variables may be **Bits**, **Bytes**, **Words**, **Dwords**, **SBytes**, **SWords**, **SDwords** or **Floats**.

Space for each variable is automatically allocated in the microcontroller's RAM area. The format for creating a variable is as follows: -

Dim Label as Size

Label is any identifier, (excluding keywords). *Size* is **Bit**, **Byte**, **Word**, **Dword**, **SByte**, **SWord**, **SDword** or **Float**. Some examples of creating variables are: -

```
Dim Cat as Bit           ' Create a single bit variable (0 or 1)
Dim Dog as Byte          ' Create an 8-bit unsigned variable (0 to 255)
Dim Rat as Word          ' Create a 16-bit unsigned variable (0 to 65535)
Dim Lrg_Rat as Dword     ' Create a 32-bit unsigned variable (0 to 4294967295)

Dim sDog as SByte        ' Create an 8-bit signed variable (-128 to +127)
Dim sRat as SWord        ' Create a 16-bit signed variable (-32768 to +32767)
Dim sLrg_Rat as SDword   ' Create a 32-bit signed variable (-2147483648 to
                          ' +2147483647)

Dim Pointy_Rat as Float  ' Create a 32-bit floating point variable
```

The number of variables available depends on the amount of RAM on a particular device and the size of the variables within the BASIC program. The compiler will reserve RAM for its own use and may also create additional temporary (System) variables for use when calculating equations, or more complex command structures. Especially if floating point calculations are carried out.

Intuitive Variable Handling.

The compiler handles its System variables intuitively, in that it only creates those that it requires. Each of the compiler's built in library subroutines i.e. **Print**, **Rsout** etc, require a certain amount of System RAM as internal variables. Previous versions of the compiler defaulted to 26 RAM spaces being created before a program had been compiled. However, with the 12-bit core device compatibility, 26 RAM slots is more than some devices possess.

Try the following program, and look at the RAM usage message on the bottom Status bar.

```
Dim Wrd1 as Word          ' Create a Word variable i.e. 16-bits
Loop:
High PORTB.0              ' Set bit 0 of PORTB high
For Wrd1= 1 to 20000 : Next ' Create a delay without using a library call
Low PORTB.0               ' Set bit 0 of PORTB high
For Wrd1= 1 to 20000 : Next ' Create a delay without using a library call
Goto Loop                 ' Do it forever
```

Only two bytes of RAM were used, and those were the ones declared in the program as variable Wrd1.

The compiler will increase its System RAM requirements as programs get larger, or more complex structures are used, such as complex expressions, inline commands used in conditions, Boolean logic used etc. However, with the limited RAM space available on some PICmicro™ devices, every byte counts.

There are certain reserved words that cannot be used as variable names, these are the system variables used by the compiler.

The following reserved words should not be used as variable names, as the compiler will create these names when required: -

PP0, PP0H, PP1, PP1H, PP2, PP2H, PP3, PP3H, PP4, PP4H, PP5, PP5H, PP6, PP6H, PP7, PP7H, PP8, PP9H, GEN, GENH, GEN2, GEN2H, GEN3, GEN3H, GEN4, GEN4H, GPR, BPF, BPFH.

RAM space required.

Each type of variable requires differing amounts of RAM memory for its allocation. The list below illustrates this.

Float	Requires 4 bytes of RAM.
Dword	Requires 4 bytes of RAM.
SDword	Requires 4 bytes of RAM.
Word	Requires 2 bytes of RAM.
SWord	Requires 2 bytes of RAM.
Byte	Requires 1 byte of RAM.
SByte	Requires 1 byte of RAM.
Bit	Requires 1 byte of RAM for every 8 Bit variables created.

Each type of variable may hold a different minimum and maximum value.

- **Bit** type variables may hold a 0 or a 1. These are created 8 at a time, therefore declaring a single **Bit** type variable in a program will not save RAM space, but it will save code space, as **Bit** type variables produce the most efficient use of code for comparisons etc.
- **Byte** type variables may hold an unsigned value from 0 to 255, and are the usual work horses of most programs. Code produced for **Byte** sized variables is very low compared to signed or unsigned **Word**, **DWord** or **Float** types, and should be chosen if the program requires faster, or more efficient operation.
- **SByte** type variables may hold a 2¹⁵ complemented signed value from -128 to +127. Code produced for **SByte** sized variables is very low compared to **SWord**, **Float**, or **SDword** types, and should be chosen if the program requires faster, or more efficient operation. However, code produced is usually larger for signed variables than unsigned types.
- **Word** type variables may hold an unsigned value from 0 to 65535, which is usually large enough for most applications. It still uses more memory than an 8-bit byte variable, but not nearly as much as a **Dword** or **SDword** type.
- **SWord** type variables may hold a 2¹⁵ complemented signed value from -32768 to +32767, which is usually large enough for most applications. **SWord** type variables will use more code space for expressions and comparisons, therefore, only use signed variables when required.

- **Dword** type variables may hold an unsigned value from 0 to 4294967295 making this the largest of the variable family types. This comes at a price however, as **Dword** calculations and comparisons will use more code space within the microcontroller. Use this type of variable sparingly, and only when necessary.
- **SDword** type variables may hold a 2³¹ complemented signed value from -2147483648 to +2147483647, also making this the largest of the variable family types. This comes at a price however, as **SDword** expressions and comparisons will use more code space than a regular **Dword** type. Use this type of variable sparingly, and only when necessary.
- **Float** type variables may theoretically hold a value from -1e37 to +1e38, but because of the 32-bit architecture of the compiler, a maximum and minimum value should be thought of as -2147483646.999 to +2147483646.999 making this the most versatile of the variable family types. However, more so than **Dword** types, this comes at a price as floating point expressions and comparisons will use more code space within the PICmicro™. Use this type of variable sparingly, and only when strictly necessary. Smaller floating point values usually offer more accuracy.

See also : Aliases, Arrays, Dim, Constants Symbol, Floating Point Math.

Floating Point Math

The Proton compiler can perform 32 x 32 bit IEEE 754 'Compliant' Floating Point calculations.

Declaring a variable as **Float** will enable floating point calculations on that variable.

```
Dim Flt as Float
```

To create a floating point constant, add a decimal point. Especially if the value is a whole number.

```
Symbol PI = 3.14 ' Create an obvious floating point constant
```

```
Symbol FlNum = 5.0 ' Create a floating point value of a whole number
```

Please note. Floating point arithmetic is not the ultimate in accuracy, it is merely a means of compressing a complex or large value into a small space (4 bytes in the compiler's case). Perfectly adequate results can usually be obtained from correct scaling of integer variables, with an increase in speed and a saving of RAM and code space. 32 bit floating point math is extremely microcontroller intensive since the PICmicro™ is only an 8 bit processor. It also consumes large amounts of RAM, and code space for its operation, therefore always use floating point sparingly, and only when strictly necessary. Floating point is not available on 12-bit core PICmicros because of memory restrictions, and is most efficient when used with 18F devices because of the more linear code and RAM specifications.

Floating Point Format

The Proton compiler uses the Microchip variation of IEEE 754 floating point format. The differences to standard IEEE 745 are minor, and well documented in Microchip application note AN575 (downloadable from www.microchip.com).

Floating point numbers are represented in a modified IEEE-754 format. This format allows the floating-point routines to take advantage of the PICmicro's architecture and reduce the amount of overhead required in the calculations. The representation is shown below compared to the IEEE-754 format: where *s* is the sign bit, *y* is the lsb of the exponent and *x* is a placeholder for the mantissa and exponent bits.

The two formats may be easily converted from one to the other by manipulation of the Exponent and Mantissa 0 bytes. The following assembly code shows an example of this operation.

Format	Exponent	Mantissa 0	Mantissa 1	Mantissa 2
IEEE-754	sxxx xxxx	yxxx xxxx	xxxx xxxx	xxxx xxxx
Microchip	xxxx xxy	sxxx xxxx	xxxx xxxx	xxxx xxxx

IEEE-754 to Microchip

```
Rlf Mantissa0  
Rlf Exponent  
Rrf Mantissa0
```

Microchip to IEEE-754

```
Rlf Mantissa0  
Rrf Exponent  
Rrf Mantissa0
```

Variables Used by the Floating Point Libraries.

Several 8-bit RAM registers are used by the math routines to hold the operands for and results of floating point operations. Since there may be two operands required for a floating point operation (such as multiplication or division), there are two sets of exponent and mantissa registers reserved (A and B). For argument A, PBP_AARGHHH holds the exponent and PBP_AARGHH, PBP_AARGH and PBP_AARG hold the mantissa. For argument B, PBP_BARGHHH holds the exponent and PBP_BARGHH, PBP_BARGH and PBP_BARG hold the mantissa.

Floating Point Example Programs.

```
' Multiply two floating point values
Device = 18F452
Declare Xtal = 4
Dim Flt as Float
Symbol FlNum = 1.234 ' Create a floating point constant value

Cls
Flt = FlNum * 10
Print Dec Flt
Stop

' Add two floating point variables
Device = 18F452
Declare Xtal = 4
Dim Flt as Float
Dim Flt1 as Float
Dim Flt2 as Float

Cls
Flt1 = 1.23
Flt2 = 1000.1
Flt = Flt1 + Flt2
Print Dec Flt
Stop

' A digital volt meter, using the on-board ADC
Device = 16F877
Declare Xtal = 4
Declare Adin_Res = 10 ' 10-bit result required
Declare Adin_Tad = FRC ' RC OSC chosen
Declare Adin_Delay = 50 ' Allow 50us sample time

Dim Raw as Word
Dim Volts as Float
Symbol Quanta = 5.0 / 1024 ' Calculate the quantising value

Cls
TRISA = %00000001 ' Configure AN0 (PORTA.0) as an input
ADCON1 = %10000000 ' Set analogue input on PORTA.0
While
Raw = Adin 0
Volts = Raw * Quanta
Print At 1,1,Dec2 Volts,"V "
Wend
```

Notes.

Any expression that contains a floating point variable or constant will be calculated as a floating point. Even if the expression also contains integer constants or variables.

If the assignment variable is an integer variable, but the expression is of a floating point nature, then the floating point result will be converted into an integer.

```
Device = 16F877
Dim Dwd as Dword
Dim Flt as Float
Symbol PI = 3.14
Flt = 10
Dwd = Flt + PI ' Float calculation will result 13.14, reduced to integer 13
Print Dec Dwd ' Display the integer result 13
Stop
```

For a more in-depth explanation of floating point, download the Microchip application notes AN575, and AN660. These can be found at www.microchip.com.

Code space requirements.

As mentioned above, floating point accuracy comes at a price of speed, and code space. Both these issues are not a problem if an 18F device is used, however 14-bit core devices can pose a problem. The compiler attempts to load the floating point libraries into low memory, along with all the other library subroutines, but if it does not fit within the first 2048 bytes of code space, and the PICmicro™ has more than 2048 bytes of code available, the floating point libraries will be loaded into the top 1000 bytes of code memory. This is invisible to the user, however, the compiler will warn that this is occurring in case that part of memory is being used by your BASIC program.

Floating Point To Integer Rounding

Assigning a floating point variable to an integer type will be truncated to the nearest value by default. For example:

```
FloatVar = 3.9
DwordVar = FloatVar
```

The variable DwordVar will hold the value of 3.

If rounding to the nearest integer value is required, use the **fRound** command.

Floating Point Exception Flags

The floating point exception flags are accessible from within the BASIC program via the system variable `_FP_FLAGS`. This must be brought into the BASIC program for the code to recognise it:

```
Dim _FP_FLAGS as Byte System
```

The exceptions are:

```
_FP_FLAGS.1      ' Floating point overflow  
_FP_FLAGS.2      ' Floating point underflow  
_FP_FLAGS.3      ' Floating point divide by zero  
_FP_FLAGS.5      ' Domain error exception
```

The exception bits can be aliased for more readability within the program:

```
Symbol FpOverflow      = _FP_FLAGS.1  ' Floating point overflow  
Symbol FpUnderFlow    = _FP_FLAGS.2  ' Floating point underflow  
Symbol FpDiv0         = _FP_FLAGS.3  ' Floating point divide by zero  
Symbol FpDomainError  = _FP_FLAGS.5  ' Domain error exception
```

After an exception is detected and handled in the program, the exception bit should be cleared so that new exceptions can be detected, however, exceptions can be ignored because new operations are not affected by old exceptions.

More Accurate Display or Conversion of Floating Point values.

By default, the compiler uses a relatively small routine for converting floating point values to decimal, ready for **Rsout**, **Print Str\$** etc. However, because of its size, it does not perform any rounding of the value first, and is only capable of converting relatively small values. i.e. approx 6 digits of accuracy. In order to produce a more accurate result, the compiler needs to use a larger routine. This is implemented by using a **Declare**: -

```
Declare Float_Display_Type = Fast or Standard
```

Using the **Fast** model for the above declare will trigger the compiler into using the more accurate floating point to decimal routine. Note that even though the routine is larger than the standard converter, it operates much faster.

The compiler defaults to Standard if the **Declare** is not issued in the BASIC program.

See also : **Dim, Symbol, Aliases, Arrays, Constants .**

Aliases

The **Symbol** directive is the primary method of creating an alias, however **Dim** can be used to create an alias to a variable. This is extremely useful for accessing the separate parts of a variable.

```
Dim Fido as Dog           ' Fido is another name for Dog
Dim Mouse as Rat.LowByte ' Mouse is the first byte (low byte) of word Rat
Dim Tail as Rat.HighByte ' Tail is the second byte (high byte) of word Rat
Dim Flea as Dog.0        ' Flea is bit-0 of Dog, which is aliased to Fido
```

There are modifiers that may also be used with variables. These are **HighByte**, **LowByte**, **Byte0**, **Byte1**, **Byte2**, **Byte3**, **Word0**, **Word1**, **SHighByte**, **SLowByte**, **SByte0**, **SByte1**, **SByte2**, **SByte3**, **SWord0**, and **SWord1**,

Word0, **Word1**, **Byte2**, **Byte3**, **SWord0**, **SWord1**, **SByte2**, and **SByte3** may only be used in conjunction with 32-bit **Dword** or **SDword** type variables.

HighByte and **Byte1** are one and the same thing, when used with a **Word** or **SWord** type variable, they refer to the unsigned High byte of a **Word** or **SWord** type variable: -

```
Dim Wrd as Word           ' Declare an unsigned Word variable
Dim Wrd_Hi as Wrd.HighByte
' Wrd_Hi now represents the unsigned high byte of variable Wrd
```

Variable **Wrd_Hi** is now accessed as a **Byte** sized type, but any reference to it actually alters the high byte of **Wrd**.

SHighByte and **SByte1** are one and the same thing, when used with a **Word** or **SWord** type variable, they refer to the signed High byte of a **Word** or **SWord** type variable: -

```
Dim Wrd as SWord          ' Declare a signed Word variable
Dim Wrd_Hi as Wrd.SHighByte
' Wrd_Hi now represents the signed high byte of variable Wrd
```

Variable **Wrd_Hi** is now accessed as an **SByte** sized type, but any reference to it actually alters the high byte of **Wrd**.

However, if **Byte1** is used in conjunction with a **Dword** type variable, it will extract the second byte. **HighByte** will still extract the high byte of the variable, as will **Byte3**. If **SByte1** is used in conjunction with an **SDword** type variable, it will extract the signed second byte. **SHighByte** will still extract the signed high byte of the variable, as will **SByte3**.

The same is true of **LowByte**, **Byte0**, **SLowByte** and **SByte0**, but they refer to the unsigned or signed Low Byte of a **Word** or **SWord** type variable: -

```
Dim Wrd as Word           ' Declare an unsigned Word variable
Dim Wrd_Lo as Wrd.LowByte
' Wrd_Lo now represents the low byte of variable Wrd
```

Variable **Wrd_Lo** is now accessed as a **Byte** sized type, but any reference to it actually alters the low byte of **Wrd**.

The modifier **Byte2** will extract the 3rd unsigned byte from a 32-bit **Dword** or **SDword** type variable as an alias. Likewise **Byte3** will extract the unsigned high byte of a 32-bit variable.

```
Dim Dwd as Dword      ' Declare a 32-bit unsigned variable named Dwd
Dim Part1 as Dwd.Byte0 ' Alias unsigned Part1 to the low byte of Dwd
Dim Part2 as Dwd.Byte1 ' Alias unsigned Part2 to the 2nd byte of Dwd
Dim Part3 as Dwd.Byte2 ' Alias unsigned Part3 to the 3rd byte of Dwd
Dim Part4 as Dwd.Byte3 ' Alias unsigned Part3 to the high (4th) byte of
```

Dwd

The modifier **SByte2** will extract the 3rd signed byte from a 32-bit **Dword** or **SDword** type variable as an alias. Likewise **SByte3** will extract the signed high byte of a 32-bit variable.

```
Dim sDwd as SDword    ' Declare a 32-bit signed variable named sDwd
Dim sPart1 as sDwd.SByte0 ' Alias signed Part1 to the low byte of sDwd
Dim sPart2 as sDwd.SByte1 ' Alias signed Part2 to the 2nd byte of sDwd
Dim sPart3 as sDwd.SByte2 ' Alias signed Part3 to the 3rd byte of sDwd
Dim sPart4 as sDwd.SByte3 ' Alias signed Part3 to the 4th byte of sDwd
```

The **Word0** and **Word1** modifiers extract the unsigned low word and high word of a **Dword** or **SDword** type variable, and is used the same as the **Byte n** modifiers.

```
Dim Dwd as Dword      ' Declare a 32-bit unsigned variable named Dwd
Dim Part1 as Dwd.Word0 ' Alias unsigned Part1 to the low word of Dwd
Dim Part2 as Dwd.Word1 ' Alias unsigned Part2 to the high word of Dwd
```

The **SWord0** and **SWord1** modifiers extract the signed low word and high word of a **Dword** or **SDword** type variable, and is used the same as the **SByte n** modifiers.

```
Dim sDwd as SDword    ' Declare a 32-bit signed variable named sDwd
Dim sPart1 as sDwd.SWord0 ' Alias Part1 to the low word of sDwd
Dim sPart2 as sDwd.SWord1 ' Alias Part2 to the high word of sDwd
```

RAM space for variables is allocated within the microcontroller in the order that they are placed in the BASIC code. For example: -

```
Dim Var1 as Byte
Dim Var2 as Byte
```

Places Var1 first, then Var2: -

```
Var1 equ n
Var2 equ n
```

This means that on a device with more than one RAM Bank, the first n variables will always be in Bank0 (the value of n depends on the specific PICmicro™ used).

Finer points for variable handling.

The position of the variable within Banks is usually of little importance if BASIC code is used, however, if assembler routines are being implemented, always assign any variables used within them first.

Problems may also arise if a **Word**, **SWord**, **Dword**, **SDword** or **Float** variable crosses a Bank boundary. If this happens, a warning message will be displayed in the error window. Most of the time, this will not cause any problems, however, to err on the side of caution, try and ensure that **Word**, **SWord**, **Dword**, **SDword** or **Float** type variables are fully inside a Bank. This is easily accomplished by placing a dummy **Byte** variable before the offending variable, or relocating the offending variable within the list of Dim statements.

Word and **SWord** type variables have a low byte and a high byte. The high byte may be accessed by simply adding the letter H to the end of the variable's name. For example: -

```
Dim Wrd as Word
```

Will produce the assembler code: -

```
Wrd equ n  
WrdH equ n
```

To access the high byte of variable Wrd, use: -

```
WrdH = 1
```

This is especially useful when assembler routines are being implemented, such as: -

```
Movlw 1  
Movwf WrdH ; Load the high byte of Wrd with 1
```

Dword, **SDWord** and **Float** type variables have a low, mid1, mid2, and high byte. The high byte may be accessed by using **Byte0**, **Byte1**, **Byte2**, or **Byte3**. For example: -

```
Dim Dwd as Dword
```

To access the high byte of variable Dwd, use: -

```
Dwd.Byte3 = 1
```

The same is true of all the alias modifiers such as **SWord0**, **Word0** etc...

Casting a variable from signed to unsigned and vice-versa is also possible using the modifiers. For example:

```
Dim sDwd as SDword ' Declare a 32-bit signed variable  
  
sDwd.Byte3 = 1 ' Load the unsigned high byte with the value 1  
sDwd.SByte0 = -1 ' Load the signed low byte with the value -1  
sDwd.SWord0 = 128 ' Load the signed low and mid1 bytes with the value 128
```

Constants

Named constants may be created in the same manner as variables. It can be more informative to use a constant name instead of a constant number. Once a constant is declared, it cannot be changed later, hence the name 'constant'.

Dim *Label as Constant expression*

```
Dim Mouse as 1
Dim Mice as Mouse * 400
Dim Mosue_PI as Mouse + 2.14
```

Although **Dim** can be used to create constants, **Symbol** is more often used.

Symbols

The **Symbol** directive provides yet another method for aliasing variables and constants. **Symbol** cannot be used to create a variable. Constants declared using **Symbol** do not use any RAM within the PICmicro™.

```
Symbol Cat = 123
Symbol Tiger = Cat           ' Tiger now holds the value of Cat
Symbol Mouse = 1             ' Same as Dim Mouse as 1
Symbol TigOuse = Tiger + Mouse ' Add Tiger to Mouse to make Tigouse
```

Floating point constants may also be created using **Symbol** by simply adding a decimal point to a value.

```
Symbol PI = 3.14           ' Create a floating point constant named PI
Symbol FlNum = 5.0         ' Create a floating point constant holding the value 5
```

Floating point constant can also be created using expressions.

```
' Create a floating point constant holding the result of the expression
Symbol Quanta = 5.0 / 1024
```

If a variable or register's name is used in a constant expression then the variable's or register's address will be substituted, not the value held in the variable or register: -

```
Symbol Con = (PORTA + 1)   ' Con will hold the value 6 (5+1)
```

Symbol is also useful for aliasing Ports and Registers: -

```
Symbol LED = PORTA.1      ' LED now references bit-1 of PORTA
Symbol TOIF = INTCON.2    ' TOIF now references bit-2 of INTCON register
```

The equal sign between the constant's name and the alias value is optional: -

```
Symbol LED PORTA.1        ' Same as Symbol LED=PORTA.1
```

Numeric Representations

The compiler recognises several different numeric representations: -

Binary is prefixed by %. i.e. **%0101**

Hexadecimal is prefixed by \$. i.e. **\$0A**

Character byte is surrounded by quotes. i.e. **"a"** represents a value of **97**

Decimal values need no prefix.

Floating point is created by using a decimal point. i.e. **3.14**

Quoted String of Characters

A Quoted String of Characters contains one or more characters (maximum 200) and is delimited by double quotes. Such as **"Hello World"**

The compiler also supports a subset of C language type formatters within a quoted string of characters. These are: -

<code>\a</code>	Bell (alert) character	\$07
<code>\b</code>	Backspace character	\$08
<code>\f</code>	Form feed character	\$0C
<code>\n</code>	New line character	\$0A
<code>\r</code>	Carriage return character	\$0D
<code>\t</code>	Horizontal tab character	\$09
<code>\v</code>	Vertical tab character	\$0B
<code>\\</code>	Backslash	\$5C
<code>\"</code>	Double quote character	\$22

Example: -

```
Print "HELLO WORLD\n\r"
```

Strings are usually treated as a list of individual character values, and are used by commands such as **Print**, **Rsout**, **Busout**, **Ewrite** etc. And of course, **String** variables.

Null Terminated

Null is a term used in computer languages for zero. So a null terminated String is a collection of characters followed by a zero in order to signify the end of characters. For example, the string of characters "Hello", would be stored as: -

```
"H", "e", "l", "l", "o", 0
```

Notice that the terminating null is the value 0 not the character "0".

Ports and other Registers

All of the PICmicro™ registers, including the ports, can be accessed just like any other byte-sized variable. This means that they can be read from, written to or used in equations directly.

```
PORTA = %01010101 ' Write value to PORTA
```

```
Var1 = Wrd * PORTA ' Multiply variable Wrd with the contents of PORTA
```

The compiler can also combine 16-bit registers such as TMR1 into a **Word** type variable. Which makes loading and reading these registers simple: -

```
' Combine TMR1L and TMR1H into unsigned Word variable wTimer1
Dim wTimer1 as TMR1L.Word

wTimer1 = 12345      ' Load TMR1L and TMR1H with the value 12345
Or
Wrd1 = wTimer1      ' Load Wrd1 with contents of TMR1
```

The **.Word** extension links registers TMR1L, and TMR1H, (which are assigned in the .ppi file associated with the relevant device used).

Any hardware register that can hold a 16-bit result can be assigned as a **Word** type variable: -

```
' Combine ADRESL and ADRESH into unsigned Word variable AD_Result
Dim AD_Result as ADRES.Word
' Combine PRODL and PRODH into unsigned Word variable MUL_PROD
Dim MUL_PROD as PRODL.Word
```

General Format

The compiler is not case sensitive, except when processing string constants such as "hello".

Multiple instructions and labels can be combined on the same line by separating them with colons ':'.
Example:

The examples below show the same program as separate lines and as a single-line: -

Multiple-line version: -

```
TRISB = %00000000      ' Make all pins on PORTB outputs
For Var1 = 0 to 100    ' Count from 0 to 100
    PORTB = Var1      ' Make PORTB = count (Var1)
Next                  ' Continue counting until 100 is reached
```

Single-line version: -

```
TRISB = %00000000 : For Var1 = 0 to 100 : PORTB = Var1 : Next
```

A Typical basic Program Layout

The compiler is very flexible, and will allow most types of constant, declaration, or variable to be placed anywhere within the BASIC program. However, it may not produce the correct results, or an unexpected syntax error may occur due to a variable being declared after it is supposed to be used.

The recommended layout for a program is shown below.

```
Device
{
  Declares
}
{
  Includes
}
{
  Constants and Variables
}

GoTo Main          ' Jump over the subroutines (if any)

{
  Subroutines go here
}
{
  Main:
  Main Program code goes here
}
```

For example:

```
Device = 18F25K20
-----
Declare Xtal = 20
Declare Hserial_Baud = 9600
-----
' Load the ADC include file (if required)
Include "ADC.inc"
-----
' Define Variables
Dim WordVar as Word          ' Create a Word size variable
-----
' Define Constants and/or aliases
Symbol Value = 10           ' Create a constant
-----
GoTo Main                    ' Jump over the subroutine/s (if any)
-----
' Simple Subroutine
AddIt:
  WordVar = WordVar + Value  ' Add the constant to the variable
  Return                    ' Return from the subroutine
-----
' Main Program Code
Main:
  WordVar = 10              ' Pre-load the variable
  GoSub AddIt              ' Call the subroutine
  Hrsout Dec WordVar, 13    ' Display the result on the serial terminal
```

Of course, it depends on what is within the include file as to where it should be placed within the program, but the above outline will usually suffice. Any include file that requires placing within a certain position within the code should be documented to state this fact.

Line Continuation Character '_'

Lines that are too long to display, may be split using the continuation character '_'. This will direct the continuation of a command to the next line. It's use is only permitted after a comma delimiter: -

```
Var1 = LookUp Var2,[1,2,3,_  
                    4,5,6,7,8]
```

OR

```
Print At 1,1,_  
"Hello World",_  
Dec Var1,_  
Hex Var2
```

Creating and using Arrays

The Proton compiler supports multi part **Byte**, **Word**, **Dword**, **SByte**, **SWord** and **SDword** variables named arrays (**Dword** and **SDword** arrays are only supported with 18F or enhanced 14-bit core devices). An array is a group of variables of the same size (8-bits, 16-bits or 32-bits wide), sharing a single name, but split into numbered cells, called elements.

An array is defined using the following syntax: -

```
Dim Name[ length ] as Byte
Dim Name[ length ] as Word
Dim Name[ length ] as Dword
Dim Name[ length ] as SByte
Dim Name[ length ] as SWord
Dim Name[ length ] as SDword
```

where *Name* is the variable's given name, and the new argument, [*length*], informs the compiler how many elements you want the array to contain. For example: -

```
Dim MyArray[10] as Byte      ' Create a 10 element unsigned byte array.
Dim MyArray[10] as Word     ' Create a 10 element unsigned word array.
Dim MyArray[10] as Dword    ' Create a 10 element unsigned dword array.
Dim sMyArray[10] as SByte   ' Create a 10 element signed byte array.
Dim sMyArray[10] as SWord   ' Create a 10 element signed word array.
Dim sMyArray[10] as SDword  ' Create a 10 element signed dword array.
```

On 18F or enhanced core devices, arrays may have as many elements as RAM permits, however, with 12-bit core and standard 14-bit core devices, arrays may contain a maximum of 256 elements, (128 for word arrays when using standard 14-bit core devices). Because of the rather complex way that some PICmicro's RAM cells are organised (i.e. Banks), there are a few rules that need to be observed when creating arrays with standard 14-bit core devices.

PICmicro™ Memory Map Complexities.

Some microcontrollers have more RAM available for variable storage, however, accessing the RAM on the standard 14-bit core devices is not as straightforward as one might expect. The RAM is organised in Banks, where each Bank is 128 bytes in length. Crossing these Banks requires bits 5 and 6 of the STATUS register to be manipulated. The larger devices such as the 16F877 have 512 RAM locations, but only 368 of these are available for variable storage, the rest are known as Special Function Registers (SFRs) and are used to control certain aspects of the microcontroller i.e. TRIS, IO ports, USART etc. The compiler attempts to make this complex system of Bank switching as transparent to the user as possible, and succeeds where standard **Bit**, **Byte**, **Word**, and **Dword** variables are concerned. However, Array variables will inevitably need to cross the Banks in order to create arrays larger than 96 bytes, which is the largest section of RAM within Bank0. Coincidentally, this is also the largest array size permissible by most other compilers at the time of writing this manual.

Large arrays (normally over 96 elements) require that their Starting address be located within the first 255 bytes of RAM (i.e. within Bank0 and Bank2), the array itself may cross this boundary. This is easily accomplished by declaring them at, or near the top of the list of variables. The compiler does not manipulate the variable declarations. If a variable is placed first in the list, it will be placed in the first available RAM slot within the microcontroller. This way, you, the programmer maintains finite control of the variable usage. For example, commonly used variables should be placed near the top of the list of declared variables. An example of declaring an array is illustrated below: -

```
Device 16F877           ' Choose a microcontroller with extra RAM
Dim Small_Array[20] as Byte ' Create a small array of 20 elements
Dim Var1 as Byte       ' Create a standard Byte variable
Dim Large_Array[256] as Byte ' Create a Byte array of 256 elements
```

OR

```
Dim Array1[120] as Byte ' Create an array of 120 elements
Dim Array2[100] as Byte ' Create another smaller array of 100 elements
```

If an array cannot be resolved, then a warning will be issued informing you of the offending line: **Warning Array 'array name' is declared at address 'array address'. Which is over the 255 RAM address limit, and crosses Bank3 boundary!**

Ignoring this warning will spell certain failure of your program.

The following array declaration will produce a warning when compiled for a 16F877 device: -

```
Device 16F877           ' Choose a microcontroller with extra RAM
Dim Array1[200] as Byte ' Create an array of 200 elements
Dim Array2[100] as Byte ' Create another smaller array of 100 elements
```

Examining the assembler code produced, will reveal that Array1 starts at address 32 and finishes at address 295. This is acceptable and the compiler will not complain. Now look at Array2, its start address is at 296 which is over the 255 address limit, thus producing a warning message.

The above warning is easily remedied by re-arranging the variable declaration list: -

```
Dim Array2[100] as Byte ' Create a small array of 100 elements
Dim Array1[200] as Byte ' Create an array of 200 elements
```

Again, examining the asm code produced, now reveals that Array2 starts at address 32 and finishes at address 163. everything OK there then. And Array1 starts at address 164 and finishes at address 427, again, its starting address was within the 255 limit so everything's OK there as well, even though the array itself crossed several Banks. A simple re-arrangement of code meant the difference between a working and not working program.

Of course, the smaller microcontrollers do not have this limitation as they do not have 255 RAM cells anyway. Therefore, arrays may be located anywhere in the variable declaration list. The same goes for the 18F devices, as these can address any area of their RAM.

18F and enhanced 14-bit core device simplicity.

The 18F devices have no such complexities in their memory map as the standard 14-bit core devices do. The memory is still banked, but each bank is 256 bytes in length, and runs linearly from one to the other. Add to that, the ability to access all RAM areas using indirect addressing, makes arrays extremely easy to use. If many large arrays are required in a program, then the 18F devices are highly recommended.

Once an array is created, its elements may be accessed numerically. Numbering starts at 0 and ends at n-1. For example: -

```
MyArray [3] = 57
Print "MyArray[3] = ", Dec MyArray[3]
```

The above example will access the fourth element in the **Byte** array and display "MyArray[3] = 57" on the LCD. The true flexibility of arrays is that the index value itself may be a variable. For example: -

```
Device 16F84           ' We'll use a smaller device this time
Dim MyArray[10] as Byte ' Create a 10-byte array.
Dim Index as Byte      ' Create a Byte variable.
For Index = 0 to 9     ' Repeat with Index= 0,1,2...9
MyArray[Index] = Index * 10 ' Write Index*10 to each element of the array.
Next
For Index = 0 to 9     ' Repeat with Index= 0,1,2...9
  Print At 1, 1, Dec MyArray [Index] ' Show the contents of each element.
  DelayMs 500          ' Wait long enough to view the values
Next
Stop
```

If the above program is run, 10 values will be displayed, counting from 0 to 90 i.e. Index * 10.

A word of caution regarding arrays: If you're familiar with other BASICs and have used their arrays, you may have run into the "subscript out of range" error. Subscript is simply another term for the index value. It is considered 'out-of range' when it exceeds the maximum value for the size of the array.

For example, in the example above, MyArray is a 10-element array. Allowable index values are 0 through 9. If your program exceeds this range, the compiler will not respond with an error message. Instead, it will access the next RAM location past the end of the array.

If you are not careful about this, it can cause all sorts of subtle anomalies, as previously loaded variables are overwritten. It's up to the programmer (you!) to help prevent this from happening.

Even more flexibility is allowed with arrays because the index value may also be an expression.

```
Device 16F84           ' We'll use a smaller device
Dim MyArray[10] as Byte ' Create a 10-byte array.
Dim Index as Byte      ' Create a Byte variable.
For Index = 0 to 8     ' Repeat with Index= 0,1,2...8
MyArray[Index + 1] = Index * 10 ' Write Index*10 to each element of array
Next
For Index = 0 to 8     ' Repeat with Index= 0,1,2...8
  Print At 1, 1, Dec MyArray[Index + 1] ' Show the contents of elements
  DelayMs 500          ' Wait long enough to view the values
Next
Stop
```

The expression within the square braces should be kept simple, and arrays are not allowed as part of the expression.

Using Arrays in Expressions.

Of course, arrays are allowed within expressions themselves. For example: -

```
Dim MyArray[10] as Byte   ' Create a 10-byte array.
Dim Index as Byte        ' Create a Byte variable.
Dim Var1 as Byte         ' Create another Byte variable
Dim Result as Byte       ' Create a variable to hold result of expression
Index = 5                ' And Index now holds the value 5
Var1 = 10                ' Variable Var1 now holds the value 10
MyArray[Index] = 20      ' Load the 6th element of MyArray with value 20
Result = (Var1 * MyArray[Index]) / 20 ' Do a simple expression
Print At 1, 1, Dec Result, " " ' Display result of expression
Stop
```

The previous example will display 10 on the LCD, because the expression reads as: -

$$(10 * 20) / 20$$

Var1 holds a value of 10, MyArray[Index] holds a value of 20, these two variables are multiplied together which will yield 200, then they're divided by the constant 20 to produce a result of 10.

An index expression used within an array that is used within an expression itself is limited to two operands.

Arrays as Strings

Arrays may also be used as simple strings in certain commands, because after all, a string is simply a byte array used to store text.

For this, the **Str** modifier is used.

The commands that support the **Str** modifier are: -

Busout - Busin
Hbusout - Hbusin
Hrsout - Hrsin
Owrite - Oread
Rsout - Rsin
Serout - Serin
Shout - Shin
Print

The **Str** modifier works in two ways, it outputs data from a pre-declared array in commands that send data i.e. **Rsout**, **Print** etc, and loads data into an array, in commands that input information i.e. **Rsin**, **Serin** etc. The following examples illustrate the **Str** modifier in each compatible command.

Using **Str** with the **Busin** and **Busout** commands.

Refer to the sections explaining the **Busin** and **Busout** commands.

Using **Str** with the **Hbusin** and **Hbusout** commands.

Refer to the sections explaining the **Hbusin** and **Hbusout** commands.

Using **Str** with the **Rsin** command.

```
Dim Array1[10] as Byte      ' Create a 10-byte array named Array1
Rsin Str Array1            ' Load 10 bytes of data directly into Array1
```

Using **Str** with the **Rsout** command.

```
Dim Array1[10] as Byte      ' Create a 10-byte array named Array1
Rsout Str Array1            ' Send 10 bytes of data directly from Array1
```

Using **Str** with the **Hrsin** and **Hrsout** commands.

Refer to the sections explaining the **Hrsout** and **Hrsin** commands.

Using **Str** with the **Shout** command.

```
Symbol DTA = PORTA.0      ' Alias the two lines for the Shout command
Symbol CLK = PORTA.1
Dim Array1[10] as Byte   ' Create a 10-byte array named Array1
' Send 10 bytes of data from Array1
Shout DTA, CLK, MSBFIRST, [Str Array1]
```

Using **Str** with the **Shin** command.

```
Symbol DTA = PORTA.0      ' Alias the two lines for the Shin command
Symbol CLK = PORTA.1
Dim Array1[10] as Byte   ' Create a 10-byte array named Array1
' Load 10 bytes of data directly into Array1
Shin DTA, CLK, MSBPRES, [Str Array1]
```

Using **Str** with the **Print** command.

```
Dim Array1[10] as Byte   ' Create a 10-byte array named Array1
Print Str Array1         ' Send 10 bytes of data directly from Array1
```

Using **Str** with the **Serout** and **Serin** commands.

Refer to the sections explaining the **Serin** and **Serout** commands.

Using **Str** with the **Oread** and **Owrite** commands.

Refer to the sections explaining the **Oread** and **Owrite** commands.

The **Str** modifier has two forms for variable-width and fixed-width data, shown below: -

Str bytearray ASCII string from bytearray until byte = 0 (null terminated).

Or array length is reached.

Str bytearray\N ASCII string consisting of N bytes from bytearray.

null terminated means that a zero (null) is placed at the end of the string of ASCII characters to signal that the string has finished.

The example below is the variable-width form of the **Str** modifier: -

```
Dim MyArray[5] as Byte   ' Create a 5 element array
MyArray[0] = "A"         ' Fill the array with ASCII
MyArray[1] = "B"
MyArray[2] = "C"
MyArray[3] = "D"
MyArray[4] = 0           ' Add the null Terminator
Print Str MyArray        ' Display the string
```

The code above displays "ABCD" on the LCD. In this form, the **Str** formatter displays each character contained in the byte array until it finds a character that is equal to 0 (value 0, not ASCII "0"). Note: If the byte array does not end with 0 (null), the compiler will read and

output all RAM register contents until it cycles through all RAM locations for the declared length of the byte array.

For example, the same code as before without a null terminator is: -

```
Dim MyArray[4] as Byte ' Create a 4 element array
MyArray[0] = "A"      ' Fill the array with ASCII
MyArray[1] = "B"
MyArray[2] = "C"
MyArray[3] = "D"
Print Str MyArray    ' Display the string
```

The code above will display the whole of the array, because the array was declared with only four elements, and each element was filled with an ASCII character i.e. "ABCD".

To specify a fixed-width format for the **Str** modifier, use the form **Str MyArray\n**; where MyArray is the byte array and n is the number of characters to display, or transmit. Changing the **Print** line in the examples above to: -

```
Print Str MyArray \ 2
```

would display "AB" on the LCD.

Str is not only used as a modifier, it is also a command, and is used for initially filling an array with data. The above examples may be re-written as: -

```
Dim MyArray[5] as Byte ' Create a 5 element array
Str MyArray = "ABCD", 0 ' Fill array with ASCII, and null terminate it
Print Str MyArray      ' Display the string
```

Strings may also be copied into other strings: -

```
Dim String1[5] as Byte ' Create a 5 element array
Dim String2[5] as Byte ' Create another 5 element array
Str String1 = "ABCD", 0 ' Fill array with ASCII, and null terminate it
Str String2 = "EFGH", 0 ' Fill other array with ASCII, null terminate it
Str String1 = Str String2 ' Copy String2 into String1
Print Str String1        ' Display the string
```

The above example will display "EFGH", because String1 has been overwritten by String2.

Using the **Str** command with **Busout**, **Hbusout**, **Shout**, and **Owrite** differs from using it with commands **Serout**, **Print**, **Hrsout**, and **Rsout** in that, the latter commands are used more for dealing with text, or ASCII data, therefore these are null terminated.

The **Hbusout**, **Busout**, **Shout**, and **Owrite** commands are not commonly used for sending ASCII data, and are more inclined to send standard 8-bit bytes. Thus, a null terminator would cut short a string of byte data, if one of the values happened to be a 0. So these commands will output data until the length of the array is reached, or a fixed length terminator is used i.e. MyArray\n.

Creating and using Strings

The Proton compiler supports **String** variables, only when targeting an 18F or enhanced 14-bit core device.

The syntax to create a string is : -

```
Dim String Name as String * String Length
```

String Name can be any valid variable name. See **Dim** .

String Length can be any value up to 255, allowing up to 255 characters to be stored.

The line of code below will create a **String** named ST that can hold 20 characters: -

```
Dim ST as String * 20
```

Two or more strings can be concatenated (linked together) by using the plus (+) operator: -

```
Device = 18F4520           ' A suitable device for Strings
' Create three strings capable of holding 20 characters
Dim DestString as String * 20
Dim SourceString1 as String * 20
Dim SourceString2 as String * 20

SourceString1 = "HELLO "   ' Load String SourceString1 with the text HELLO
' Load String SourceString2 with the text WORLD
SourceString2 = "WORLD"
' Add both Source Strings together. Place result into String DestString
DestString = SourceString1 + SourceString2
```

The String DestString now contains the text "HELLO WORLD", and can be transmitted serially or displayed on an LCD: -

```
Print DestString
```

The Destination String itself can be added to if it is placed as one of the variables in the addition expression. For example, the above code could be written as: -

```
Device = 18F452           ' A suitable device for Strings
' Create a String capable of holding 20 characters
Dim DestString as String * 20
' Create another String capable of holding 20 characters
Dim SourceString as String * 20

DestString = "HELLO "     ' Pre-load String DestString with the text HELLO
SourceString = "WORLD"    ' Load String SourceString with the text WORLD
' Concatenate DestString with SourceString
DestString = DestString + SourceString
Print DestString         ' Display the result which is "HELLO WORLD"
Stop
```

Note that Strings cannot be subtracted, multiplied or divided, and cannot be used as part of a regular expression otherwise a syntax error will be produced.

It's not only other strings that can be added to a string, the functions **Cstr**, **Estr**, **Mid\$**, **Left\$**, **Right\$**, **Str\$**, **ToUpper**, and **ToLower** can also be used as one of variables to concatenate.

A few examples of using these functions are shown below: -

Cstr Example

```
' Use Cstr function to place a code memory string into a RAM String variable

Device = 18F4520           ' A suitable device for Strings
' Create a String capable of holding 20 characters
Dim DestString as String * 20
Dim SourceString as String * 20 ' Create another String
SourceString = "HELLO "      ' Load the string with characters
DestString = SourceString + Cstr CodeStr ' Concatenate the string
Print DestString            ' Display the result which is "HELLO WORLD"
Stop
CodeStr:
  Cdata "WORLD",0
```

The above example is really only for demonstration because if a Label name is placed as one of the parameters in a string concatenation, an automatic (more efficient) **Cstr** operation will be carried out. Therefore the above example should be written as: -

More efficient Example of above code

```
' Place a code memory string into a String variable more efficiently than
' using Cstr

Device = 18F4520           ' A suitable device for Strings
' Create a String capable of holding 20 characters
Dim DestString as String * 20
Dim SourceString as String * 20 ' Create another String
SourceString = "HELLO "      ' Load the string with characters
DestString = SourceString + CodeStr ' Concatenate the string
Print DestString            ' Display the result which is "HELLO WORLD"
Stop
CodeStr:
  Cdata "WORLD",0
```

A null terminated string of characters held in Data (on-board eeprom) memory can also be loaded or concatenated to a string by using the **Estr** function: -

Estr Example

```
' Use the Estr function in order to place a
' Data memory string into a String variable
' Remember to place Edata before the main code
' so it's recognised as a constant value

Device = 18F4520           ' A suitable device for Strings
Dim DestString as String * 20 ' Create a String for 20 characters
Dim SourceString as String * 20 ' Create another String

Data_Str Edata "WORLD",0    ' Create a string in Data memory
SourceString = "HELLO "     ' Load the string with characters
DestString = SourceString + Estr Data_Str ' Concatenate the strings
Print DestString            ' Display the result which is "HELLO WORLD"
Stop
```


Converting an integer or floating point value into a string is accomplished by using the **Str\$** function: -

Str\$ Example

```
' Use the Str$ function in order to concatenate
' an integer value into a String variable

Device = 18F4520           ' A suitable device for Strings
Dim DestString as String * 30 ' Create a String capable of holding 30
characters
Dim SourceString as String * 20 ' Create another String
Dim Wrld1 as Word           ' Create a Word variable

Wrld1 = 1234               ' Load the Word variable with a value
SourceString = "Value = " ' Load the string with characters
DestString = SourceString + Str$(Dec Wrld1) ' Concatenate the string
Print DestString          ' Display the result which is "Value = 1234"
Stop
```

Left\$ Example

```
' Copy 5 characters from the left of SourceString
' and add to a quoted character string

Device = 18F4520           ' A suitable device for Strings
Dim SourceString as String * 20 ' Create a String
Dim DestString as String * 20 ' Create another String

SourceString = "HELLO WORLD" ' Load the source string with characters
DestString = Left$(SourceString, 5) + " WORLD"
Print DestString            ' Display the result which is "HELLO WORLD"
Stop
```

Right\$ Example

```
' Copy 5 characters from the right of SourceString
' and add to a quoted character string

Device = 18F4520           ' A suitable device for Strings
Dim SourceString as String * 20 ' Create a String
Dim DestString as String * 20 ' Create another String

SourceString = "HELLO WORLD" ' Load the source string with characters
DestString = "HELLO " + Right$(SourceString, 5)
Print DestString            ' Display the result which is "HELLO WORLD"
Stop
```

Mid\$ Example

```
' Copy 5 characters from position 4 of SourceString
' and add to quoted character strings

Device = 18F4520           ' A suitable device for Strings
Dim SourceString as String * 20 ' Create a String
Dim DestString as String * 20 ' Create another String

SourceString = "HELLO WORLD" ' Load the source string with characters
DestString = "HEL" + Mid$(SourceString, 4, 5) + "RLD"
Print DestString            ' Display the result which is "HELLO WORLD"
Stop
```

Converting a string into uppercase or lowercase is accomplished by the functions **ToUpper** and **ToLower**: -

ToUpper Example

```
' Convert the characters in SourceString to upper case

Device = 18F4520           ' A suitable device for Strings
Dim SourceString as String * 20 ' Create a String
Dim DestString as String * 20  ' Create another String

SourceString = "hello world" ' Load source with lowercase characters
DestString = ToUpper(SourceString)
Print DestString           ' Display the result which is "HELLO WORLD"
Stop
```

ToLower Example

```
' Convert the characters in SourceString to lower case

Device = 18F4520           ' A suitable device for Strings
Dim SourceString as String * 20 ' Create a String
Dim DestString as String * 20  ' Create another String

SourceString = "HELLO WORLD" ' Load the string with uppercase characters
DestString = ToLower(SourceString)
Print DestString           ' Display the result which is "hello world"
Stop
```

Loading a String Indirectly

If the Source String is assigned or unsigned **Byte**, **Word**, **Float** or an **Array** variable, the value contained within the variable is used as a pointer to the start of the Source String's address in RAM.

Example

```
' Copy SourceString into DestString using a pointer to SourceString

Device = 18F4520           ' A suitable device for Strings
Dim SourceString as String * 20 ' Create a String
Dim DestString as String * 20  ' Create another String
' Create a Word variable to hold the address of SourceString
Dim StringAddr as Word

SourceString = "HELLO WORLD" ' Load the source string with characters
' Locate the start address of SourceString in RAM
StringAddr = AddressOf(SourceString)
DestString = StringAddr      ' Source string into the destination string
Print DestString           ' Display the result, which will be "HELLO"
Stop
```

Slicing a String.

Each position within the string can be accessed the same as an unsigned **Byte Array** by using square braces: -

```
Device = 18F4520          ' A suitable device for Strings
Dim SourceString as String * 20 ' Create a String

SourceString[0] = "H"    ' Place letter "H" as first character in the string
SourceString[1] = "E"    ' Place the letter "E" as the second character
SourceString[2] = "L"    ' Place the letter "L" as the third character
SourceString[3] = "L"    ' Place the letter "L" as the fourth character
SourceString[4] = "O"    ' Place the letter "O" as the fifth character
SourceString[5] = 0      ' Add a null to terminate the string

Print SourceString      ' Display the string, which will be "HELLO"
Stop
```

The example above demonstrates the ability to place individual characters anywhere in the string. Of course, you wouldn't use the code above in an actual BASIC program.

A string can also be read character by character by using the same method as shown above: -

```
Device = 18F4520          ' A suitable device for Strings
Dim SourceString as String * 20 ' Create a String
Dim Var1 as Byte

SourceString = "HELLO"    ' Load the source string with characters
' Copy character 1 from the source string and place it into Var1
Var1 = SourceString[1]
Print Var1 ' Display character extracted from string. Which will be "E"
Stop
```

When using the above method of reading and writing to a string variable, the first character in the string is referenced at 0 onwards, just like an unsigned **Byte Array**.

The example below shows a more practical String slicing demonstration.

```
' Display a string's text by examining each character individually
Device = 18F4520          ' A suitable device for Strings
Dim SourceString as String * 20 ' Create a String
Dim Charpos as Byte      ' Holds the position within the string

SourceString = "HELLO WORLD" ' Load the source string with characters
Charpos = 0              ' Start at position 0 within the string
Repeat                   ' Create a loop
  ' Display the character extracted from the string
  Print SourceString[Charpos]
  Inc Charpos             ' Move to the next position within the string
  ' Keep looping until the end of the string is found
Until Charpos = Len(SourceString)
Stop
```

Notes

A word of caution regarding Strings: If you're familiar with interpreted BASICs and have used their String variables, you may have run into the "subscript out of range" error. This error occurs when the amount of characters placed in the string exceeds its maximum size.

For example, in the examples above, most of the strings are capable of holding 20 characters. If your program exceeds this range by trying to place 21 characters into a string only created for 20 characters, the compiler will not respond with an error message. Instead, it will access the next RAM location past the end of the String.

If you are not careful about this, it can cause all sorts of subtle anomalies as previously loaded variables are overwritten. It's up to the programmer (you!) to prevent this from happening by ensuring that the **String** in question is large enough to accommodate all the characters required, but not too large that it uses up too much precious RAM.

The compiler will help by giving a reminder message when appropriate, but this can be ignored if you are confident that the **String** is large enough.

See also : **Creating and using Virtual Strings with Cdata**
Creating and using Virtual Strings with Edata
Cdata, Len, Left\$, Mid\$, Right\$
String Comparisons, Str\$, ToLower, ToUpper, AddressOf.

Creating and using Code Memory Strings

Some devices have the ability to read and write to their own flash memory. And although writing to this memory too many times is unhealthy for the PICmicro™, reading this memory is both fast, and harmless. Which offers a unique form of data storage and retrieval, the **Cdata** command and the new **Dim** as Code directive proves this, as they uses the mechanism of reading and storing in the microcontroller's flash memory.

Combining the unique features of the 'self modifying devices ' with a string format, the compiler is capable of reducing the overhead of printing, or transmitting large amounts of text data. The **Cstr** modifier may be used in commands that deal with text processing i.e. **Print**, **Serout**, **Hrsout**, and **Rsout** .

The **Cstr** modifier is used in conjunction with the **Cdata** command. The **Cdata** command is used for initially creating the string of characters: -

```
String1: Cdata "HELLO WORLD", 0
```

The above line of code will create, in flash memory, the values that make up the ASCII text "HELLO WORLD", at address String1. Note the null terminator after the ASCII text.

null terminated means that a zero (null) is placed at the end of the string of ASCII characters to signal that the string has finished.

To display, or transmit this string of characters, the following command structure could be used:

```
Print Cstr String1
```

The label that declared the address where the list of **Cdata** values resided, now becomes the string's name. In a large program with lots of text formatting, this type of structure can save quite literally hundreds of bytes of valuable code space.

Try both these small programs, and you'll see that using **Cstr** saves a few bytes of code: -

First the standard way of displaying text: -

```
Device = 18F4520
Cls
Print "HELLO WORLD"
Print "HOW ARE YOU?"
Print "I AM FINE!"
Stop
```

Now using the **Cstr** modifier: -

```
Cls
Print Cstr TEXT1
Print Cstr TEXT2
Print Cstr TEXT3
Stop
```

```
TEXT1: Cdata "HELLO WORLD", 0
TEXT2: Cdata "HOW ARE YOU?", 0
TEXT3: Cdata "I AM FINE!", 0
```

Again, note the null terminators after the ASCII text in the **Cdata** commands. Without these, the microcontroller will continue to transmit data in an endless loop.

The term 'virtual string' relates to the fact that a string formed from the **Cdata** command cannot (rather should not) be written too, but only read from.

Not only label names can be used with the **Cstr** modifier, constants, variables and expressions can also be used that will hold the address of the **Cdata** 's label (a pointer). For example, the program below uses a **Word** size variable to hold 2 pointers (address of a label, variable or array) to 2 individual null terminated text strings formed by **Cdata** .

Example 1

```
' Use the Proton development board for the example
Include "Proton_4.Inc"
Dim Address as Word      ' Pointer variable

DelayMs 100              ' Wait for the LCD to stabilise
Cls                      ' Clear the LCD

Address = String1       ' Point address to string 1
Print Cstr Address      ' Display string 1
Address = String2       ' Point Address to string 2
Print Cstr Address      ' Display string 2
Stop

' Create the text to display
String1:
  Cdata "Hello ", 0
String2:
  Cdata "World", 0
```

It is also possible to eliminate the **Cstr** modifier altogether and place the label's name directly. The compiler will see this as an implied **Cstr** and act accordingly. For example:

```
' Use the Proton development board for the example
Include "Proton18_4.Inc"

Dim CodeString1 as Code = "Hello ", 0
Dim CodeString2 as Code = "World", 0

Cls                      ' Clear the LCD

Print CodeString1       ' Display CodeString1
Print CodeString2       ' Display CodeString2
Stop
```

Creating and using Eeprom Memory Strings with Edata

Some 14-bit core and most 18F microcontrollers have on-board eeprom memory, and although writing to this memory too many times is unhealthy for the device, reading this memory is both fast and harmless. Which offers a great place for text storage and retrieval.

Combining the eeprom memory of a device with a string format, the compiler is capable of reducing the overhead of printing, or transmitting large amounts of text data using a memory resource that is very often left unused and ignored. The **Estr** modifier may be used in commands that deal with text processing i.e. **Print**, **Serout**, **Hrsout**, and **Rsout** and **String** handling etc.

The **Estr** modifier is used in conjunction with the **Edata** command, which is used to initially create the string of characters: -

```
String1 Edata "HELLO WORLD", 0
```

The above line of code will create, in eeprom memory, the values that make up the ASCII text "HELLO WORLD", at address String1 in Data memory. Note the null terminator after the ASCII text.

To display, or transmit this string of characters, the following command structure could be used:

```
Print Estr String1
```

The identifier that declared the address where the list of **Edata** values resided, now becomes the string's name. In a large program with lots of text formatting, this type of structure can save many bytes of valuable code space.

Try both these small programs, and you'll see that using **Estr** saves code space: -

First the standard way of displaying text: -

```
Device 18F4520
Cls
Print "HELLO WORLD"
Print "HOW ARE YOU?"
Print "I AM FINE!"
Stop
```

Now using the **Estr** modifier: -

```
TEXT1 Edata "HELLO WORLD", 0
TEXT2 Edata "HOW ARE YOU?", 0
TEXT3 Edata "I AM FINE!", 0

Cls
Print Estr TEXT1
Print Estr TEXT2
Print Estr TEXT3
Stop
```

Again, note the null terminators after the ASCII text in the **Edata** commands. Without these, the PICmicro™ will continue to transmit data in an endless loop.

The term 'virtual string' relates to the fact that a string formed from the **Edata** command cannot (rather should not) be written to often, but can be read as many times as wished without causing harm to the device.

Not only identifiers can be used with the **Estr** modifier, constants, variables and expressions can also be used that will hold the address of the **Edata**'s identifier (a pointer). For example, the program below uses a **Byte** size variable to hold 2 pointers (address of a variable or array) to 2 individual null terminated text strings formed by **Edata** .

```
' Use the Proton development board for the example
  Include "Proton_4.Inc"

  Dim Address as Word ' Pointer variable
' Create the text to display in eeprom memory
String1 Edata "HELLO ", 0
String2 Edata "WORLD", 0

  DelayMs 100          ' Wait for the LCD to stabilise
  Cls                  ' Clear the LCD
  Address = String1    ' Point address to string 1
  Print Estr Address   ' Display string 1
  Address = String2    ' Point Address to string 2
  Print Estr Address   ' Display string 2
  Stop
```

Notes

Note that the identifying text *must* be located on the same line as the **Edata** directive or a syntax error will be produced. It must also not contain a postfix colon as does a line label or it will be treat as a line label. Think of it as an alias name to a constant.

Any **Edata** directives *must* be placed at the head of the BASIC program as is done with **Symbols**, so that the name is recognised by the rest of the program as it is parsed. There is no need to jump over **Edata** directives as you have to with **Ldata** or **Cdata**, because they do not occupy code memory, but reside in high Data memory.

String Comparisons

Just like any other variable type, **String** variables can be used within comparisons such as **If-Then**, **Repeat-Until**, and **While-Wend**. In fact, it's an essential element of any programming language. However, there are a few rules to obey because of the PICmicro's architecture.

Equal (=) or Not Equal (<>) comparisons are the only type that apply to Strings, because one **String** can only ever be equal or not equal to another **String**. It would be unusual (unless your using the C language) to compare if one **String** was greater or less than another.

So a valid comparison could look something like the lines of code below: -

```
If String1 = String2 Then Print "EQUAL" : Else : Print "not EQUAL"
or
If String1 <> String2 Then Print "not EQUAL" : Else : Print "EQUAL"
```

But as you've found out if you read the *Creating Strings* section, there is more than one type of **String** in a PICmicro™. There is a **String** variable, a code memory string, and a quoted character string.

Note that pointers to **String** variables are not allowed in comparisons, and a syntax error will be produced if attempted.

Starting with the simplest of string comparisons, where one string variable is compared to another string variable. The line of code would look similar to either of the two lines above.

Example 1

' Simple string variable comparison

```
Device = 18F452           ' A suitable device for Strings
' Create a String capable of holding 20 characters
Dim String1 as String * 20
Dim String2 as String * 20 ' Create another String

Cls
String1 = "EGGS"          ' Pre-load String String1 with the text EGGS
String2 = "BACON"         ' Load String String2 with the text BACON

If String1 = String2 Then ' Is String1 equal to String2 ?
    Print At 1,1, "EQUAL" ' Yes. So display EQUAL on line 1 of the LCD
Else                       ' Otherwise
    Print At 1,1, "not EQUAL" ' Display not EQUAL on line 1 of the LCD
EndIf

String2 = "EGGS"          ' Now make the strings the same as each other
If String1 = String2 Then ' Is String1 equal to String2 ?
    Print At 2,1, "EQUAL"  ' Yes. So display EQUAL on line 2 of the LCD
Else                       ' Otherwise
    Print At 2,1, "not EQUAL" ' Display not EQUAL on line 2 of the LCD
EndIf
Stop
```

The example above will display not Equal on line one of the LCD because String1 contains the text "EGGS" while String2 contains the text "BACON", so they are clearly not equal.

Line two of the LCD will show Equal because String2 is then loaded with the text "EGGS" which is the same as String1, therefore the comparison is equal.

A similar example to the previous one uses a quoted character string instead of one of the **String** variables.

Example 2

```
' String variable to Quoted character string comparison

Device = 18F4520          ' A suitable device for Strings
' Create a String capable of holding 20 characters
Dim String1 as String * 20

Cls
String1 = "EGGS"          ' Pre-load String String1 with the text EGGS

If String1 = "BACON" Then ' Is String1 equal to "BACON" ?
    Print At 1,1, "equal" ' Yes. So display EQUAL on line 1 of the LCD
Else                       ' Otherwise...
    Print At 1,1, "not equal" ' Display not EQUAL on line 1 of the LCD
EndIf

If String1 = "EGGS" Then  ' Is String1 equal to "EGGS" ?
    Print At 2,1, "equal"  ' Yes. So display EQUAL on line 2 of the LCD
Else                       ' Otherwise...
    Print At 2,1, "not equal" ' Display not EQUAL on line 2 of the LCD
EndIf
Stop
```

The example above produces exactly the same results as example1 because the first comparison is clearly not equal, while the second comparison is equal.

Example 3

```
' Use a string comparison in a Repeat-Until loop

Device = 18F4520          ' A suitable device for Strings
Dim SourceString as String * 20 ' Create a String
Dim DestString as String * 20   ' Create another String
Dim Charpos as Byte           ' Character position within the strings

Cls
Clear DestString           ' Fill DestString with nulls
SourceString = "HELLO"     ' Load String SourceString with the text HELLO

Repeat                     ' Create a loop
    ' Copy SourceString into DestString one character at a time
    DestString[Charpos] = SourceString[Charpos]
    Inc Charpos             ' Move to the next character in the strings
    ' Stop when DestString is equal to the text "HELLO"
Until DestString = "HELLO"
Print DestString           ' Display DestString
Stop
```

Example 4

```
' Compare a string variable to a string held in code memory
Device = 18F4520          ' A suitable device for Strings
' Create a String capable of holding 20 characters
Dim String1 as String * 20

Cls
String1 = "BACON"        ' Pre-load String String1 with the text BACON
If CodeString= "BACON" Then ' Is CodeString equal to "BACON" ?
    Print At 1,1, " equal " ' Yes. So display EQUAL on line 1 of the LCD
Else                       ' Otherwise...
    Print At 1,1, "not equal" ' Display not EQUAL on line 1 of the LCD
EndIf

String1 = "EGGS"        ' Pre-load String String1 with the text EGGS
If String1 = CodeString Then ' Is String1 equal to CodeString ?
    Print At 2,1, " equal " ' Yes. So display EQUAL on line 2 of the LCD
Else                       ' Otherwise...
    Print At 2,1, "not equal " ' Display not EQUAL on line 2 of the LCD
EndIf
Stop

CodeString:
Cdata "EGGS", 0
```

Example 5

```
' String comparisons using Select-Case
Device = 18F4520          ' A suitable device for Strings
' Create a String capable of holding 20 characters
Dim String1 as String * 20

Cls
String1 = "EGGS"        ' Pre-load String String1 with the text EGGS
Select String1          ' Start comparing the string
Case "EGGS"             ' Is String1 equal to EGGS?
    Print At 1,1,"Found EGGS"
Case "BACON"           ' Is String1 equal to BACON?
    Print At 1,1,"Found BACON"
Case "COFFEE"          ' Is String1 equal to COFFEE?
    Print At 1,1,"Found COFFEE"
Case Else               ' Default to...
    Print At 1,1,"No Match" ' Displaying no match
EndSelect
Stop
```

See also : **Creating and using Strings**
Creating and using Virtual Strings with Cdata
Cdata, If-Then-Else-EndIf, Repeat-Until
Select-Case, While-Wend .

Relational Operators

Relational operators are used to compare two values. The result can be used to make a decision regarding program flow.

The list below shows the valid relational operators accepted by the compiler:

Operator	Relation	Expression Type
=	Equality	$X = Y$
==	Equality	$X == Y$ (Same as above Equality)
<>	Inequality	$X <> Y$
!=	Inequality	$X != Y$ (Same as above Inequality)
<	Less than	$X < Y$
>	Greater than	$X > Y$
<=	Less than or Equal to	$X <= Y$
>=	Greater than or Equal to	$X >= Y$

See also : **If-Then-Else-EndIf, Repeat-Until, Select-Case, While-Wend.**

Boolean Logic Operators

The **If-Then-Else-EndIf**, **While-Wend**, and **Repeat-Until** conditions now support the logical operators **and** and **or**.

The operators **and** and **or** join the results of two conditions to produce a single true/false result. **and** and **or** work the same as they do in everyday speech. Run the example below once with **and** (as shown) and again, substituting **or** for **and**: -

```
Dim Var1 as Byte
Dim Var2 as Byte
Cls
Var1 = 5
Var2 = 9
If Var1 = 5 and Var2 = 10 Then Res_True
Stop
Res_True:
Print "Result Is True."
Stop
```

The condition "Var1 = 5 **and** Var2 = 10" is not true. Although Var1 is 5, Var2 is not 10. **and** works just as it does in plain English, both conditions must be true for the statement to be true. **or** also works in a familiar way; if one or the other or both conditions are true, then the statement is true. **xor** (short for exclusive-or) may not be familiar, but it does have an English counterpart: If one condition or the other (but not both) is true, then the statement is true.

Parenthesis (or rather the lack of it!).

Every compiler has its quirky rules, and the Proton compiler is no exception. One of its quirks means that parenthesis is not supported in a Boolean condition, or indeed with any of the **If-Then-Else-EndIf**, **While-Wend**, and **Repeat-Until** conditions. Parenthesis in an expression within a condition is allowed however. So, for example, the expression: -

```
If (Var1 + 3) = 10 Then do something.    Is allowed.
but: -
If( (Var1 + 3) = 10) Then do something.  Is not allowed.
```

The boolean operands do have a precedence within a condition. The **and** operand has the highest priority, then the **or**, then the **xor**. This means that a condition such as: -

```
If Var1 = 2 and Var2 = 3 or Var3 = 4 Then do something
```

Will compare Var1 and Var2 to see if the **and** condition is true. It will then see if the **or** condition is true, based on the result of the **and** condition.

Then operand always required.

The Proton compiler relies heavily on the **Then** part. Therefore, if the **Then** part of a condition is left out of the code listing, a *Syntax Error* will be produced.

Math Operators

The Proton compiler performs all math operations in full hierarchical order. Which means that there is precedence to the operators. For example, multiplies and divides are performed before adds and subtracts. To ensure the operations are carried out in the correct order use parenthesis to group the operations: -

$$A = ((B - C) * (D + E)) / F$$

All math operations are signed or unsigned depending on the variable type used, and performed with 16, or 32-bit or floating point precision, again, depending on the variable types and constant values used within the expression.

The operators supported are: -

Addition '+'.	Adds variables and/or constants.
Subtraction '-'.	Subtracts variables and/or constants.
Multiply '*'.	Multiplies variables and/or constants.
Multiply High '***'.	Returns the high 16 bits of an unsigned 16-bit integer multiply.
Multiply Middle '*/'.	Returns the middle 16 bits of an unsigned 16-bit integer multiply.
Divide '/'.	Divides variables and/or constants.
Modulus '//.	Returns the remainder after dividing one integer value by another.
Bitwise and '&'.	Returns the logical AND of two values.
Bitwise or ' '.	Returns the logical OR of two values.
Bitwise xor '^'.	Returns the logical XOR of two values.
Bitwise Shift Left '<<'.	Shifts the bits of a value left a specified number of places.
Bitwise Shift Right '>>'.	Shifts the bits of a value right a specified number of places.
Bitwise Complement '~'.	Reverses the bits in a variable.
Bitwise Reverse '@'.	Reverses the order of the lowest bits in a value.
Abs.	Returns the absolute value of a signed number.
Acos	Returns the Arc Cosine of a floating point value in radians.
Asin	Returns the Arc Sine of a floating point value in radians.
Atan	Returns the Arc Tangent of a floating point value in radians.
Cos.	Returns the Cosine of a floating point value in radians.
Dcd.	2 n -power decoder of a four-bit value.
Decimal Digit Extract '?'.	Returns the specified decimal digit of a positive value.
Exp	Deduce the exponential function of a floating point value.
ISin	Returns the Sine of an integer value in radians.
ICos	Returns the Cosine of an integer value in radians.
Isqr	Returns the Square Root of an integer value.
Log	Returns the Natureal Log of a floating point value.
Log10	Returns the Log of a floating point value.
Ncd.	Priority encoder of a 16-bit value.
Pow	Computes a variable to the power of another.
Sin.	Returns the Sine of a floating point value in radians.
Sqr.	Returns the Square Root of a floating point value.
Tan	Returns the Tangent of a floating point value in radians.
Div32.	15-bit x 31 bit unsigned divide. (For PBP compatibility only)

Add '+'

Syntax

Assignment Variable = Variable + Variable

Overview

Adds variables and/or constants, returning an unsigned or signed 8, 16, 32-bit or floating point result.

Operators

Assignment Variable can be any valid variable type.

Variable can be a constant, variable or expression.

Addition works exactly as you would expect with signed and unsigned integers as well as floating point.

```
Dim Value1 as Word
Dim Value2 as Word
Value1 = 1575
Value2 = 976
Value1 = Value1 + Value2      ' Add the numbers.
Print Dec Value1             ' Display the result

' 32-bit addition
Dim Value1 as Word
Dim Value2 as Dword
Value1 = 1575
Value2 = 9763647
Value2 = Value2 + Value1     ' Add the numbers.
Print Dec Value1            ' Display the result
```

Subtract '-'

Syntax

Assignment Variable = Variable - Variable

Overview

Subtracts variables and/or constants, returning an unsigned or signed 8, 16, 32-bit or floating point result.

Operators

Assignment Variable can be any valid variable type.

Variable can be a constant, variable or expression.

Subtract works exactly as you would expect with signed and unsigned integers as well as floating point.

```
Dim Value1 as Word
Dim Value2 as Word
Value1 = 1000
Value2 = 999
Value1 = Value1 - Value2     ' Subtract the numbers.
Print Dec Value1            ' Display the result
```

```
' 32-bit subtraction
Dim Value1 as Word
Dim Value2 as Dword
Value1 = 1575
Value2 = 9763647
Value2 = Value2 - Value1      ' Subtract the numbers.
Print Dec Value1             ' Display the result

' 32-bit signed subtraction
Dim Value1 as Dword
Dim Value2 as Dword
Value1 = 1575
Value2 = 9763647
Value1 = Value1 - Value2     ' Subtract the numbers.
Print Sdec Value1           ' Display the result
```

Multiply '*'

Syntax

*Assignment Variable = Variable * Variable*

Overview

Multiplies variables and/or constants, returning an unsigned or signed 8, 16, 32-bit or floating point result.

Operators

Assignment Variable can be any valid variable type.

Variable can be a constant, variable or expression.

Multiply works exactly as you would expect with signed or unsigned integers from -2147483648 to +2147483647 as well as floating point. If the result of multiplication is larger than 2147483647 when using 32-bit variables, the excess bit will be lost.

```
Dim Value1 as Word
Dim Value2 as Word
Value1 = 1000
Value2 = 19
Value1 = Value1 * Value2     ' Multiply Value1 by Value2.
Print Dec Value1            ' Display the result

' 32-bit multiplication
Dim Value1 as Word
Dim Value2 as Dword
Value1 = 100
Value2 = 10000
Value2 = Value2 * Value1     ' Multiply the numbers.
Print Dec Value1            ' Display the result
```


Multiply High '**'

Syntax

*Assignment Variable = Variable ** Variable*

Overview

Multiplies 8 or 16-bit unsigned variables and/or constants, returning the high 16 bits of the result.

Operators

Assignment Variable can be any valid variable type.

Variable can be a constant, variable or expression.

When multiplying two 16-bit values, the result can be as large as 32 bits. Since the largest variable supported by the compiler is 16-bits, the highest 16 bits of a 32-bit multiplication result are normally lost. The ** (double-star) operand produces these upper 16 bits.

For example, suppose 65000 (\$FDE8) is multiplied by itself. The result is 4,225,000,000 or \$FBD46240. The * (star, or normal multiplication) instruction would return the lower 16 bits, \$6240. The ** instruction returns \$FBD4.

```
Dim Value1 as Word
Dim Value2 as Word
Value1 = $FDE8
Value2 = Value1 ** Value1      ' Multiply $FDE8 by itself
Print Hex Value2              ' Return high 16 bits.
```

Notes.

This operand enables compatibility with BASIC STAMP code, and melab's compiler code, but is rather obsolete considering the 32-bit capabilities of the Proton compiler.

Multiply Middle '*/'

Syntax

*Assignment Variable = Variable */ Variable*

Overview

Multiplies unsigned variables and/or constants, returning the middle 16 bits of the 32-bit result.

Operators

Assignment Variable can be any valid variable type.

Variable can be a constant, variable or expression.

The Multiply Middle operator (*/) has the effect of multiplying a value by a whole number and a fraction. The whole number is the upper byte of the multiplier (0 to 255 whole units) and the fraction is the lower byte of the multiplier (0 to 255 units of 1/256 each). The */ operand allows a workaround for the compiler's integer-only math.

Suppose we are required to multiply a value by 1.5. The whole number, and therefore the upper byte of the multiplier, would be 1, and the lower byte (fractional part) would be 128, since $128/256 = 0.5$. It may be clearer to express the */ multiplier in Hex as \$0180, since hex keeps the contents of the upper and lower bytes separate. Here's an example: -

```
Dim Value1 as Word
Value1 = 100
Value1 = Value1 */ $0180      ' Multiply by 1.5 [1 + (128/256)]
Print Dec Value1             ' Display result (150).
```

To calculate constants for use with the */ instruction, put the whole number portion in the upper byte, then use the following formula for the value of the lower byte: -

$$\text{int}(\text{fraction} * 256)$$

For example, take Pi (3.14159). The upper byte would be \$03 (the whole number), and the lower would be $\text{int}(0.14159 * 256) = 36$ (\$24). So the constant Pi for use with */ would be \$0324. This isn't a perfect match for Pi, but the error is only about 0.1%.

Notes.

This operand enables compatibility with BASIC STAMP code, and melab's compiler code, but is rather obsolete considering the 32-bit capabilities of the Proton compiler.

Divide '/'

Syntax

Assignment Variable = Variable / Variable

Overview

Divides variables and/or constants, returning an unsigned or signed 8, 16, 32-bit or floating point result.

Operators

Assignment Variable can be any valid variable type.

Variable can be a constant, variable or expression.

The Divide operator (/) works exactly as you would expect with signed or unsigned integers from -2147483648 to +2147483647 as well as floating point.

```
Dim Value1 as Word
Dim Value2 as Word
Value1 = 1000
Value2 = 5
Value1 = Value1 / Value2      ' Divide the numbers.
Print Dec Value1             ' Display the result (200).
```

' 32-bit division

```
Dim Value1 as Word
Dim Value2 as Dword
Value1 = 100
Value2 = 10000
Value2 = Value2 / Value1      ' Divide the numbers.
Print Dec Value1             ' Display the result
```

Integer Modulus '//'

Syntax

Assignment Variable = Variable // Variable

Overview

Return the remainder left after dividing one unsigned or signed value by another.

Operators

Assignment Variable can be any valid variable type.

Variable can be a constant, variable or expression.

Some division problems don't have a whole-number result; they return a whole number and a fraction. For example, $1000/6 = 166.667$. Integer math doesn't allow the fractional portion of the result, so $1000/6 = 166$. However, 166 is an approximate answer, because $166*6 = 996$. The division operation left a remainder of 4. The // returns the remainder of a given division operation. Numbers that divide evenly, such as $1000/5$, produce a remainder of 0: -

```
Dim Value1 as Word
Dim Value2 as Word
Value1 = 1000
Value2 = 6
Value1 = Value1 // Value2      ' Get remainder of Value1 / Value2.
Print Dec Value1              ' Display the result (4).

' 32-bit modulus
Dim Value1 as Word
Dim Value2 as Dword
Value1 = 100
Value2 = 99999
Value2 = Value2 // Value1      ' mod the numbers.
Print Dec Value1              ' Display the result
```

The modulus operator does not operate with floating point values or variables.

Logical and '&'

The And operator (&) returns the bitwise and of two values. Each bit of the values is subject to the following logic: -

0 and 0 = 0
0 and 1 = 0
1 and 0 = 0
1 and 1 = 1

The result returned by & will contain 1s in only those bit positions in which both input values contain 1s: -

```
Dim Value1 as Byte
Dim Value2 as Byte
Dim Result as Byte
Value1 = %00001111
Value2 = %10101101
Result = Value1 & Value2
Print Bin Result      ' Display and result (%00001101)
```

OR

```
Print Bin ( %00001111 & %10101101 ) ' Display and result (%00001101)
```

Bitwise operations are not permissible with floating point values or variables.

Logical or '|'

The Or operator (|) returns the bitwise or of two values. Each bit of the values is subject to the following logic: -

0 or 0 = 0
0 or 1 = 1
1 or 0 = 1
1 or 1 = 1

The result returned by | will contain 1s in any bit positions in which one or the other (or both) input values contain 1s: -

```
Dim Value1 as Byte
Dim Value2 as Byte
Dim Result as Byte
Value1 = %00001111
Value2 = %10101001
Result = Value1 | Value2
Print Bin Result      ' Display or result (%10101111)
```

OR

```
Print Bin ( %00001111 | %10101001 ) ' Display or result (%10101111)
```

Bitwise operations are not permissible with floating point values or variables.

Logical Xor '^'

The Xor operator (^) returns the bitwise xor of two values. Each bit of the values is subject to the following logic: -

```
0 xor 0 = 0
0 xor 1 = 1
1 xor 0 = 1
1 xor 1 = 0
```

The result returned by ^ will contain 1s in any bit positions in which one or the other (but not both) input values contain 1s: -

```
Dim Value1 as Byte
Dim Value2 as Byte
Dim Result as Byte
Value1 = %00001111
Value2 = %10101001
Result = Value1 ^ Value2
Print Bin Result           ' Display xor result (%10100110)
```

or

```
Print Bin ( %00001111 ^ %10101001 ) ' Display xor result (%10100110)
```

Bitwise operations are not permissible with floating point values or variables.

Bitwise Shift Left '<<'

Shifts the bits of a value to the left a specified number of places. Bits shifted off the left end of a number are lost; bits shifted into the right end of the number are 0s. Shifting the bits of a value left *n* number of times also has the effect of multiplying that number by two to the *n*th power.

For example $100 \ll 3$ (shift the bits of the decimal number 100 left three places) is equivalent to $100 * 2^3$.

```
Dim Value1 as Word
Dim Loop as Byte
Value1 = %1111111111111111
For Loop = 1 to 16           ' Repeat with b0 = 1 to 16.
    Print Bin Value1 << Loop ' Shift Value1 left Loop places.
Next
```

Bitwise operations are not permissible with floating point values or variables. All bit shifts are unsigned, regardless of the variable type used.

Bitwise Shift Right '>>'

Shifts the bits of a variable to the right a specified number of places. Bits shifted off the right end of a number are lost; bits shifted into the left end of the number are 0s. Shifting the bits of a value right n number of times also has the effect of dividing that number by two to the n th power.

For example $100 \gg 3$ (shift the bits of the decimal number 100 right three places) is equivalent to $100 / 2^3$.

```
Dim Value1 as Word
Dim Loop as Byte
Value1 = %1111111111111111
For Loop = 1 to 16          ' Repeat with b0 = 1 to 16.
    Print Bin Value1 >> Loop  ' Shift Value1 right Loop places.
Next
```

Complement '~'

The Complement operator (~) inverts the bits of a value. Each bit that contains a 1 is changed to 0 and each bit containing 0 is changed to 1. This process is also known as a "bitwise not".

```
Dim Value1 as Word
Dim Value2 as Word
Value2 = %1111000011110000
Value1 = ~Value2          ' Complement Value2.
Print Bin16 Value1       ' Display the result
```

Complementing can be carried out with all variable types except **Floats**. Attempting to complement a floating point variable will produce a syntax error. All bit shifts are unsigned, regardless of the variable type used.

Bitwise Reverse '@'

Reverses the order of the lowest bits in a value. The number of bits to be reversed is from 1 to 32. Its syntax is: -

```
MyVar = %10101100 @ 4      ' Sets MyVar to %10100011
OR
Dim MyDword as Dword
' Sets MyDword to %10101010000000001111111110100011
MyDword = %10101010000000001111111110101100 @ 4
```

Decimal Digit extract '?'

In this form, the ? operator is compatible with the BASIC Stamp, and the melab's PicBASIC Pro compiler. It returns the specified decimal digit of a 16-bit positive value. Digits are numbered from 0 (the rightmost digit) to 4 (the leftmost digit of a 16-bit number; 0 to 65535). Example: -

```
MyWord = 9742
Print MyWord ? 2          ' Display digit 2 (7)
For Loop = 0 to 4
    Print MyWord ? Loop   ' Display digits 0 through 4 of 9742.
Next
```

Note

Decimal Digit Extract does not support **Float** type variables.

Abs

Syntax

Assignment Variable = **Abs**(*Variable*)

Overview

Return the absolute value of a constant, variable or expression.

Operators

Assignment Variable can be any valid variable type.

Variable can be a constant, variable or expression.

32-bit Example

```
Device = 18F4520
Dim Dwd1 as Dword ' Declare an unsigned Dword variable
Dim Dwd2 as Dword ' Declare an unsigned Dword variable
Cls
Dwd1 = -1234567 ' Load Dwd1 with value -1234567
Dwd2 = Abs(Dwd1) ' Extract the absolute value from Dwd1
Print Dec Dwd2 ' Display the result, which is 1234567
Stop
```

Floating Point example

```
Device = 18F4520
Dim Flp1 as Float ' Declare a Float variable
Dim Flp2 as Float ' Declare a Float variable
Cls
Flp1 = -1234567 ' Load Flp1 with value -1234567
Flp2 = Abs(Flp1) ' Extract the absolute value from Flp1
Print Dec Flp2 ' Display the result, which is 1234567
Stop
```

Note

When implementing trigonometry, or other built in, functions within an expression, always wrap them in parenthesis, otherwise the parser may consider the extra operands as part of the trigonometry parameter and produce an incorrect result. For example:

```
MyAssignment = (Abs(MyVar1)) + MyVar2
```

fAbs

Syntax

Assignment Variable = **fAbs**(*Variable*)

Overview

Return the absolute value of a constant, variable or expression as floating point.

Operators

Assignment Variable can be any valid variable type.

Variable can be a constant, variable or expression.

Floating Point example

```
Device = 18F4520
Dim Flp1 as Float ' Declare a Float variable
Dim Flp2 as Float ' Declare a Float variable
Cls
Flp1 = -1234567 ' Load Flp1 with value -1234567
Flp2 = fAbs(Flp1) ' Extract the absolute value from Flp1
Print Dec Flp2 ' Display the result, which is 1234567
Stop
```

Note

When implementing trigonometry, or other built in, functions within an expression, always wrap them in parenthesis, otherwise the parser may consider the extra operands as part of the trigonometry parameter and produce an incorrect result. For example:

```
MyAssignment = (fAbs(MyVar1)) + MyVar2
```


Acos

Syntax

Assignment Variable = **Acos**(Variable)

Overview

Deduce the Arc Cosine of a floating point value

Operators

Assignment Variable can be any valid variable type.

Variable can be a constant, variable or expression that requires the Arc Cosine (Inverse Cosine) extracted. The value expected and returned by the floating point **Acos** is in radians. The value must be in the range of -1 to +1

Example

```
Include "Proton18_4.Inc" ' Use the Proton board for the demo

Dim Floatin as Float    ' Holds the value to Acos
Dim Floatout as Float   ' Holds the result of the Acos
DelayMs 100             ' Wait for the LCD to stabilise
Cls                     ' Clear the LCD
Floatin = 0.8           ' Load the variable
Floatout = Acos(Floatin) ' Extract the Acos of the value
Print Dec Floatout      ' Display the result
Stop
```

Notes

Acos is not implemented with 12, or 14-bit core devices, however, with the extra functionality, and more linear memory offered by the 18F devices, full 32-bit floating point Arc Cosine is implemented.

Floating point trigonometry is extremely memory hungry, so do not be surprised if a large chunk of the microcontroller's code memory is used with a single operator. This also means that floating point trigonometry is comparatively slow to operate.

When implementing trigonometry, or other built in, functions within an expression, always wrap them in parenthesis, otherwise the parser may consider the extra operands as part of the trigonometry parameter and produce an incorrect result. For example:

```
MyAssignment = (Acos(MyVar1)) + MyVar2
```

Asin

Syntax

Assignment Variable = **Asin**(Variable)

Overview

Deduce the Arc Sine of a floating point value

Operators

Assignment Variable can be any valid variable type.

Variable can be a constant, variable or expression that requires the Arc Sine (Inverse Sine) extracted. The value expected and returned by **Asin** is in radians. The value must be in the range of -1 to +1

Example

```
Include "Proton18_4.Inc" ' Use the Proton board for the demo

Dim Floatin as Float    ' Holds the value to Asin
Dim Floatout as Float   ' Holds the result of the Asin
DelayMs 100             ' Wait for the LCD to stabilise
Cls                     ' Clear the LCD
Floatin = 0.8           ' Load the variable
Floatout = Asin(Floatin) ' Extract the Asin of the value
Print Dec Floatout      ' Display the result
Stop
```

Notes

Asin is not implemented with 12, or 14-bit core devices, however, with the extra functionality, and more linear memory offered by the 18F devices, full 32-bit floating point Arc Sine is implemented.

Floating point trigonometry is extremely memory hungry, so do not be surprised if a large chunk of the microcontroller's code memory is used with a single operator. This also means that floating point trigonometry is comparatively slow to operate.

When implementing trigonometry, or other built in, functions within an expression, always wrap them in parenthesis, otherwise the parser may consider the extra operands as part of the trigonometry parameter and produce an incorrect result. For example:

```
MyAssignment = (Asin(MyVar1)) + MyVar2
```

Atan

Syntax

Assignment Variable = **Atan**(*Variable*)

Overview

Deduce the Arc Tangent of a floating point value

Operators

Assignment Variable can be any valid variable type.

Variable can be a constant, variable or expression that requires the Arc Tangent (Inverse Tangent) extracted. The value expected and returned by the floating point **Atan** is in radians.

Example

```
Include "Proton18_4.Inc" ' Use the Proton board for the demo
Dim Floatin as Float    ' Holds the value to Atan
Dim Floatout as Float   ' Holds the result of the Atan
DelayMs 100             ' Wait for the LCD to stabilise
Cls                     ' Clear the LCD
Floatin = 1             ' Load the variable
Floatout = Atan(Floatin) ' Extract the Atan of the value
Print Dec Floatout     ' Display the result
Stop
```

Notes

Atan is not implemented with 12, or 14-bit core devices, however, with the extra functionality, and more linear memory offered by the 18F devices, full 32-bit floating point Arc Tangent is implemented.

Floating point trigonometry is extremely memory hungry, so do not be surprised if a large chunk of the microcontroller's code memory is used with a single operator. This also means that floating point trigonometry is comparatively slow to operate.

When implementing trigonometry, or other built in, functions within an expression, always wrap them in parenthesis, otherwise the parser may consider the extra operands as part of the trigonometry parameter and produce an incorrect result. For example:

```
MyAssignment = (Atan(MyVar1)) + MyVar2
```

Cos

Syntax

Assignment Variable = **Cos**(*Variable*)

Overview

Deduce the Cosine of a floating point value

Operators

Assignment Variable can be any valid variable type.

Variable can be a constant, variable or expression that requires the Cosine extracted. The value expected and returned by **Cos** is in radians.

Example

```
Include "Proton18_4.Inc" ' Use the Proton board for the demo
Dim Floatin as Float    ' Holds the value to Cos with
Dim Floatout as Float   ' Holds the result of the Cos
DelayMs 100             ' Wait for the LCD to stabilise
Cls                     ' Clear the LCD
Floatin = 123           ' Load the variable
Floatout = Cos(Floatin) ' Extract the Cos of the value
Print Dec Floatout     ' Display the result
Stop
```

Notes

With 12, and 14-bit core devices, **Cos** returns the 8-bit cosine of a value, compatible with the BASIC Stamp syntax. The result is in two's complement form (i.e. -127 to 127). **Cos** starts with a value in binary radians, 0 to 255, instead of the customary 0 to 359 degrees.

However, with the extra functionality, and more linear memory offered by the 18F devices, full 32-bit floating point Cosine is implemented.

Floating point trigonometry is extremely memory hungry, so do not be surprised if a large chunk of the microcontroller's code memory is used with a single operator. This also means that floating point trigonometry is comparatively slow to operate.

When implementing trigonometry, or other built in, functions within an expression, always wrap them in parenthesis, otherwise the parser may consider the extra operands as part of the trigonometry parameter and produce an incorrect result. For example:

```
MyAssignment = (Cos(MyVar1)) + MyVar2
```

Dcd

2ⁿ-power decoder of a four-bit value. **Dcd** accepts a value from 0 to 15, and returns a 16-bit number with that bit number set to 1. For example: -

```
Wrd1= Dcd 12      ' Set bit 12.  
Print Bin16 Wrd1 ' Display result (%0001000000000000)
```

Dcd does not support **Dword**, or **Float** type variables. Therefore the highest value obtainable is 65535.

Exp

Syntax

Assignment Variable = **Exp**(*Variable*)

Overview

Deduce the exponential function of a floating point value. This is e to the power of *value* where e is the base of natural logarithms. **Exp** 1 is 2.7182818.

Operators

Assignment Variable can be any valid variable type.

Variable can be a constant, variable or expression.

Example

```
Include "Proton18_4.Inc" ' Use the Proton board for the demo
Dim Floatin as Float    ' Holds the value to Exp with
Dim Floatout as Float   ' Holds the result of the Exp
DelayMs 100             ' Wait for the LCD to stabilise
Cls                     ' Clear the LCD
Floatin = 1             ' Load the variable
Floatout = Exp(Floatin) ' Extract the Exp of the value
Print Dec Floatout     ' Display the result
Stop
```

Notes

Exp is not implemented with 12, or 14-bit core devices, however, with the extra functionality, and more linear memory offered by the 18F devices, full 32-bit floating point exponentials are implemented.

Floating point trigonometry is extremely memory hungry, so do not be surprised if a large chunk of the microcontroller's code memory is used with a single operator. This also means that floating point trigonometry is comparatively slow to operate.

When implementing trigonometry, or other built in, functions within an expression, always wrap them in parenthesis, otherwise the parser may consider the extra operands as part of the trigonometry parameter and produce an incorrect result. For example:

```
MyAssignment = (Exp(MyVar1)) + MyVar2
```

fRound

Syntax

Assignment Variable = **fRound**(*Variable*)

Overview

Round a value, variable or expression to the nearest integer.

Operators

Assignment Variable can be any valid variable type.

Variable can be a constant, variable or expression.

Example

```
Include "Proton18_4.Inc"      ' Use the Proton board for the demo
Dim Floatin as Float         ' Holds the value to round
Dim Dwordout as Dword        ' Holds the result of fRound
DelayMs 100                   ' Wait for the LCD to stabilise
Cls                            ' Clear the LCD
Floatin = 1.9                 ' Load the variable
Dwordout = fRound(Floatin)    ' Round to the nearest integer value
Print Dec Dwordout           ' Display the integer result
Stop
```

Notes

Floating point trigonometry is extremely memory hungry, so do not be surprised if a large chunk of the microcontroller's code memory is used with a single operator. This also means that floating point trigonometry is comparatively slow to operate.

When implementing trigonometry, or other built in, functions within an expression, always wrap them in parenthesis, otherwise the parser may consider the extra operands as part of the trigonometry parameter and produce an incorrect result. For example:

```
MyAssignment = (fRound(MyVar1)) + MyVar2
```

ISin

Syntax

Assignment Variable = **ISin**(*Variable*)

Overview

Deduce the integer Sine of an integer value

Operators

Assignment Variable can be any valid variable type.

Variable can be a constant, variable or expression that requires the Sine extracted. The value expected and returned by **ISin** is in decimal radians (0 to 255).

Example

```
Include "Proton18_4.Inc" ' Use the Proton board for the demo
Dim ByteIn as Byte      ' Holds the value to Sin
Dim ByteOut as Byte     ' Holds the result of the Sin
DelayMs 100             ' Wait for the LCD to stabilise
Cls                     ' Clear the LCD
ByteIn = 123            ' Load the variable
ByteOut = ISin(ByteIn) ' Extract the integer Sin of the value
Print Dec ByteOut      ' Display the result
Stop
```

Note

When implementing trigonometry, or other built in, functions within an expression, always wrap them in parenthesis, otherwise the parser may consider the extra operands as part of the trigonometry parameter and produce an incorrect result. For example:

```
MyAssignment = (ISin(MyVar1)) + MyVar2
```


ICos

Syntax

Assignment Variable = **ICos**(*Variable*)

Overview

Deduce the integer Cosine of an integer value

Operators

Assignment Variable can be any valid variable type.

Variable can be a constant, variable or expression that requires the Cosine extracted. The value expected and returned by **ICos** is in decimal radians (0 to 255).

Example

```
Include "Proton18_4.Inc" ' Use the Proton board for the demo
Dim ByteIn as Byte      ' Holds the value to Cos
Dim ByteOut as Byte     ' Holds the result of the Cos
DelayMs 100             ' Wait for the LCD to stabilise
Cls                     ' Clear the LCD
ByteIn = 123            ' Load the variable
ByteOut = ICos(ByteIn) ' Extract the integr Cosine of the value
Print Dec ByteOut      ' Display the result
Stop
```

Note

When implementing trigonometry, or other built in, functions within an expression, always wrap them in parenthesis, otherwise the parser may consider the extra operands as part of the trigonometry parameter and produce an incorrect result. For example:

```
MyAssignment = (ICos(MyVar1)) + MyVar2
```

Isqr

Syntax

Assignment Variable = **ISqr**(*Variable*)

Overview

Deduce the integer Square Root of an integer value

Operators

Assignment Variable can be any valid variable type.

Variable can be a constant, variable or expression that requires the Square Root extracted.

Example

```
Include "Proton18_4.Inc" ' Use the Proton board for the demo
Dim ByteIn as Byte      ' Holds the value to Sqr
Dim ByteOut as Byte     ' Holds the result of the Sqr
DelayMs 100             ' Wait for the LCD to stabilise
Cls                     ' Clear the LCD
ByteIn = 123            ' Load the variable
ByteOut = ISqr(ByteIn) ' Extract the integr square root of the value
Print Dec ByteOut      ' Display the result
Stop
```

Note

When implementing trigonometry, or other built in, functions within an expression, always wrap them in parenthesis, otherwise the parser may consider the extra operands as part of the trigonometry parameter and produce an incorrect result. For example:

```
MyAssignment = (ISqr(MyVar1)) + MyVar2
```

Log

Syntax

Assignment Variable = **Log**(*Variable*)

Overview

Deduce the Natural Logarithm a floating point value

Operators

Assignment Variable can be any valid variable type.

Variable can be a constant, variable or expression that requires the natural logarithm extracted.

Example

```
Include "Proton18_4.Inc" ' Use the Proton board for the demo
Dim Floatin as Float    ' Holds the value to Log with
Dim Floatout as Float   ' Holds the result of the Log
DelayMs 100             ' Wait for the LCD to stabilise
Cls                     ' Clear the LCD
Floatin = 1             ' Load the variable
Floatout = Log(Floatin) ' Extract the Log of the value
Print Dec Floatout     ' Display the result
Stop
```

Notes

Log is not implemented with 12, or 14-bit core devices, however, with the extra functionality, and more linear memory offered by the 18F devices, full 32-bit floating point Natural Logarithms are implemented.

Floating point trigonometry is extremely memory hungry, so do not be surprised if a large chunk of the microcontroller's code memory is used with a single operator. This also means that floating point trigonometry is comparatively slow to operate.

When implementing trigonometry, or other built in, functions within an expression, always wrap them in parenthesis, otherwise the parser may consider the extra operands as part of the trigonometry parameter and produce an incorrect result. For example:

```
MyAssignment = (Log(MyVar1)) + MyVar2
```

Log10

Syntax

Assignment Variable = **Log10**(*Variable*)

Overview

Deduce the Logarithm a floating point value

Operators

Assignment Variable can be any valid variable type.

Variable can be a constant, variable or expression that requires the Logarithm extracted.

Example

```
Include "Proton18_4.Inc" ' Use the Proton board for the demo
Dim Floatin as Float    ' Holds the value to Log10 with
Dim Floatout as Float   ' Holds the result of the Log10
DelayMs 100             ' Wait for the LCD to stabilise
Cls                     ' Clear the LCD
Floatin = 1             ' Load the variable
Floatout = Log10(Floatin) ' Extract the Log10 of the value
Print Dec Floatout      ' Display the result
Stop
```

Notes

Log10 is not implemented with 12, or 14-bit core devices, however, with the extra functionality, and more linear memory offered by the 18F devices, full 32-bit floating point logarithms are implemented.

Floating point trigonometry is extremely memory hungry, so do not be surprised if a large chunk of the microcontroller's code memory is used with a single operator. This also means that floating point trigonometry is comparatively slow to operate.

When implementing trigonometry, or other built in, functions within an expression, always wrap them in parenthesis, otherwise the parser may consider the extra operands as part of the trigonometry parameter and produce an incorrect result. For example:

```
MyAssignment = (Log10(MyVar1)) + MyVar2
```

Ncd

Priority encoder of a 16-bit value. Ncd takes a 16-bit value, finds the highest bit containing a 1 and returns the bit position plus one (1 through 16). If no bit is set, the input value is 0. Ncd returns 0. Ncd is a fast way to get an answer to the question "what is the largest power of two that this value is greater than or equal to?" The answer that Ncd returns will be that power, plus one. Example: -

```
Wrd1= %1101          ' Highest bit set is bit 3.  
Print Dec Ncd Wrd1  ' Display the Ncd of Wrd1(4).
```

Ncd does not support **Dword**, or **Float** type variables.

Pow

Syntax

Assignment Variable = **Pow**(*Variable*, *Pow Variable*)

Overview

Computes *Variable* to the power of *Pow Variable*.

Operators

Assignment Variable can be any valid variable type.

Variable can be a constant, variable or expression.

Pow Variable can be a constant, variable or expression.

Example

```
Include "Proton18_4.Inc" ' Use the Proton board for the demo
Dim PowOf as Float
Dim Floatin as Float ' Holds the value to Pow with
Dim Floatout as Float ' Holds the result of the Pow
DelayMs 100 ' Wait for the LCD to stabilise
Cls ' Clear the LCD
PowOf= 10
Floatin = 2 ' Load the variable
Floatout = Pow(Floatin,PowOf) ' Extract the Pow of the value
Print Dec Floatout ' Display the result
Stop
```

Notes

Pow is not implemented with 12, or 14-bit core devices, however, with the extra functionality, and more linear memory offered by the 18F devices, full 32-bit floating point power of is implemented.

Floating point trigonometry is extremely memory hungry, so do not be surprised if a large chunk of the microcontroller's code memory is used with a single operator. This also means that floating point trigonometry is comparatively slow to operate.

When implementing trigonometry, or other built in, functions within an expression, always wrap them in parenthesis, otherwise the parser may consider the extra operands as part of the trigonometry parameter and produce an incorrect result. For example:

```
MyAssignment = (Pow(MyVar1, MyVar2)) + MyVar3
```

Sin

Syntax

Assignment Variable = **Sin**(*Variable*)

Overview

Deduce the Sine of a floating point value

Operators

Assignment Variable can be any valid variable type.

Variable can be a constant, variable or expression that requires the Sine extracted. The value expected and returned by **Sin** is in radians.

Example

```
Include "Proton18_4.Inc" ' Use the Proton board for the demo
Dim Floatin as Float    ' Holds the value to Sin
Dim Floatout as Float   ' Holds the result of the Sin
DelayMs 100             ' Wait for the LCD to stabilise
Cls                     ' Clear the LCD
Floatin = 123           ' Load the variable
Floatout = Sin(Floatin) ' Extract the Sin of the value
Print Dec Floatout      ' Display the result
Stop
```

Notes

With 12, and 14-bit core devices, **Sin** returns the 8-bit sine of a value, compatible with the BASIC Stamp syntax. The result is in two's complement form (i.e. -127 to 127). **Sin** starts with a value in binary radians, 0 to 255, instead of the customary 0 to 359 degrees.

However, with the extra functionality, and more linear memory offered by the 18F devices, full 32-bit floating point Sine is implemented.

Floating point trigonometry is extremely memory hungry, so do not be surprised if a large chunk of the microcontroller's code memory is used with a single operator. This also means that floating point trigonometry is comparatively slow to operate.

When implementing trigonometry, or other built in, functions within an expression, always wrap them in parenthesis, otherwise the parser may consider the extra operands as part of the trigonometry parameter and produce an incorrect result. For example:

```
MyAssignment = (Sin(MyVar1)) + MyVar2
```

Sqr

Syntax

Assignment Variable = **Sqr**(*Variable*)

Overview

Deduce the Square Root of a floating point value

Operators

Assignment Variable can be any valid variable type.

Variable can be a constant, variable or expression that requires the Square Root extracted.

Notes

With 12 and 14-bit core devices, **Sqr** returns an integer square root of a value, compatible with the BASIC Stamp syntax. Remember that most square roots have a fractional part that the compiler discards in doing its integer-only math. Therefore it computes the square root of 100 as 10 (correct), but the square root of 99 as 9 (the actual is close to 9.95). Example: -

```
Var1 = Sqr Var2
```

or

```
Print Sqr 100      ' Display square root of 100 (10).  
Print Sqr 99      ' Display of square root of 99 (9 due to truncation)
```

However, with the extra functionality, and more linear memory offered by the 18F devices, full 32-bit floating point **Sqr** is implemented.

Example

```
Include "Proton18_4.Inc"  ' Use the Proton board for the demo  
Dim Floatin as Float   ' Holds the value to Sqr  
Dim Floatout as Float  ' Holds the result of the Sqr  
DelayMs 100             ' Wait for the LCD to stabilise  
Cls                    ' Clear the LCD  
Floatin = 600            ' Load the variable  
Floatout = Sqr(Floatin)  ' Extract the Sqr of the value  
Print Dec Floatout      ' Display the result  
Stop
```

Notes

Floating point trigonometry is extremely memory hungry, so do not be surprised if a large chunk of the microcontroller's code memory is used with a single operator. This also means that floating point trigonometry is comparatively slow to operate.

When implementing trigonometry, or other built in, functions within an expression, always wrap them in parenthesis, otherwise the parser may consider the extra operands as part of the trigonometry parameter and produce an incorrect result. For example:

```
MyAssignment = (Sqr(MyVar1)) + MyVar2
```


Tan

Syntax

Assignment Variable = **Tan**(*Variable*)

Overview

Deduce the Tangent of a floating point value

Operators

Assignment Variable can be any valid variable type.

Variable can be a constant, variable or expression that requires the Tangent extracted. The value expected and returned by the floating point **Tan** is in radians.

Example

```
Include "Proton18_4.Inc" ' Use the Proton board for the demo
Dim Floatin as Float    ' Holds the value to Tan
Dim Floatout as Float   ' Holds the result of the Tan
DelayMs 100             ' Wait for the LCD to stabilise
Cls                     ' Clear the LCD
Floatin = 1             ' Load the variable
Floatout = Tan(Floatin) ' Extract the Tan of the value
Print Dec Floatout     ' Display the result
Stop
```

Notes

Tan is not implemented with 12, or 14-bit core devices, however, with the extra functionality, and more linear memory offered by the 18F devices, full 32-bit floating point tangent is implemented.

Floating point trigonometry is extremely memory hungry, so do not be surprised if a large chunk of the microcontroller's code memory is used with a single operator. This also means that floating point trigonometry is comparatively slow to operate.

When implementing trigonometry, or other built in, functions within an expression, always wrap them in parenthesis, otherwise the parser may consider the extra operands as part of the trigonometry parameter and produce an incorrect result. For example:

```
MyAssignment = (Tan(MyVar1)) + MyVar2
```

Div32

In order to make the Proton compiler more compatible with code produced for the melab's PicBASIC Pro compiler, the **Div32** operator has been added. The melab's compiler's multiply operand operates as a 16-bit x 16-bit multiply, thus producing a 32-bit result. However, since the compiler only supports a maximum variable size of 16 bits (**Word**), access to the result had to happen in 2 stages: -

```
Var = Var1 * Var2 returns the lower 16 bits of the multiply
```

while...

```
Var = Var1 ** Var2 returns the upper 16 bits of the multiply
```

There was no way to access the 32-bit result as a valid single value.

In many cases it is desirable to be able to divide the entire 32-bit result of the multiply by a 16-bit number for averaging, or scaling. **Div32** is actually limited to dividing a 31-bit unsigned integer (0 - 2147483647) by a 15-bit unsigned integer (0 - 32767). This ought to be sufficient in most situations.

Because the melab's compiler only allows a maximum variable size of 16 bits (0 - 65535), **Div32** relies on the fact that a multiply was performed just prior to the **Div32** command, and that the internal compiler variables still contain the 32-bit result of the multiply. No other operation may occur between the multiply and the **Div32** or the internal variables may be altered, thus destroying the 32-bit multiplication result.

The following example demonstrates the operation of **Div32**: -

```
Dim Wrd1 as Word
Dim Wrd2 as Word
Dim Wrd3 as Word
Dim Fake as Word      ' Must be a Word type variable for result

Wrd2 = 300
Wrd3 = 1000

Fake = Wrd2 * Wrd3     ' Operators ** or */ could also be used instead
Wrd1 = Div32 100
Print Dec Wrd1
```

The above program assigns Wrd2 the value 300 and Wrd3 the value 1000. When multiplied together, the result is 300000. However, this number exceeds the 16-bit word size of a variable (65535). Therefore, the dummy variable, Fake, contains only the lower 16 bits of the result. **Div32** uses the compiler's internal (System) variables as the operands.

Note.

This operand enables a certain compatibility with melab's compiler code, but is rather obsolete considering the 32-bit, and floating point capabilities of the Proton compiler.

Commands and Directives

Adin	Read the on-board analogue to digital converter.
Asm-EndAsm	Insert assembly language code section.
Box	Draw a square on a graphic LCD.
Branch	Computed Goto (equiv. to On..Goto).
BranchL	Branch out of page (long Branch).
Break	Exit a loop prematurely.
Bstart	Send a Start condition to the I ² C bus.
Bstop	Send a Stop condition to the I ² C bus.
Brestart	Send a Restart condition to the I ² C bus.
BusAck	Send an Acknowledge condition to the I ² C bus.
BusNack	Send an Not Acknowledge condition to the I ² C bus.
Busin	Read bytes from an I ² C device.
Busout	Write bytes to an I ² C device.
Button	Detect and debounce a key press.
Call	Call an assembly language subroutine.
Cdata	Define initial contents in memory.
CF_Init	Initialise the interface to a Compact Flash card.
CF_Sector	Point to the sector of interest in a Compact Flash card.
CF_Read	Read data from a Compact Flash card.
CF_Write	Write data to a Compact Flash card.
Circle	Draw a circle on a graphic LCD.
Clear	Place a variable or bit in a low state, or clear all RAM area.
ClearBit	Clear a bit of a port or variable, using a variable index.
Cls	Clear the LCD.
Config	Set or Reset programming fuse configurations.
Continue	Cause the next iteration of the enclosing loop to begin.
Counter	Count the number of pulses occurring on a pin.
Cread	Read data from code memory.
Cread8, Cread16, Cread32	Read a single or multi-byte value from an Cdata table with more efficiency than Cread .
Cursor	Position the cursor on the LCD.
Cwrite	Write data to code memory.
Dec	Decrement a variable.
Declare	Adjust library routine parameters.
DelayMs	Delay (1 millisecond resolution).
DelayUs	Delay (1 microsecond resolution).
Device	Choose the type of PICmicro™ to compile with.
Dig	Return the value of a decimal digit.
Dim	Create a variable.
Disable	Disable software interrupts previously Enabled.
DTMFout	Produce a DTMF Touch Tone note.
Edata	Define initial contents of on-board eeprom.
Enable	Enable software interrupts previously Disabled.
End	Stop execution of the BASIC program.
Eread	Read a value from on-board eeprom.
Ewrite	Write a value to on-board eeprom.
For...to...Next...Step	Repeatedly execute statements.
Freqout	Generate one or two tones, of differing or the same frequencies.
GetBit	Examine a bit of a port or variable, using a variable index.
Gosub	Call a BASIC subroutine at a specified label.
Goto	Continue execution at a specified label.
HbStart	Send a Start condition to the I ² C bus using the MSSP module.
HbStop	Send a Stop condition to the I ² C bus using the MSSP module.

HbRestart	Send a Restart condition to the I ² C bus using the MSSP module.
HbusAck	Send an Ack condition to the I ² C bus using the MSSP module.
HbusNack	Send a Not Ack condition to the I ² C bus using the MSSP module.
Hbusin	Read from an I ² C device using the MSSP module.
Hbusout	Write to an I ² C device using the MSSP module.
High	Make a pin or port high.
Hpwm	Generate a Pwm signal using the CCP module.
Hrsin	Receive data from the serial port on devices that contain a USART.
Hrsout	Transmit data from the serial port on devices that contain a USART.
Hserin	Receive data from the serial port on devices that contain a USART.
Hserout	Transmit data from the serial port on devices that contain a USART.
Hrsin2	Same as Hrsin but using a 2nd USART if available.
Hrsout2	Same as Hrsout but using a 2nd USART if available.
Hserin2	Same as Hserin but using a 2nd USART if available.
Hserout2	Same as Hserout but using a 2nd USART if available.
I2Cin	Read bytes from an I ² C device with user definable SDA\SCL lines.
I2Cout	Write bytes to an I ² C device with user definable SDA\SCL lines.
If..Then..Elseif..Else..Endif	Conditionally execute statements.
Inc	Increment a variable.
Include	Load a BASIC file into the source code.
Inkey	Scan a keypad.
Input	Make pin an input.
LCDread	Read a single byte from a Graphic LCD.
LCDwrite	Write bytes to a Graphic LCD.
Left\$	Extract <i>n</i> amount of characters from the left of a String.
Ldata	Place information into code memory. For access by Lread .
Line	Draw a line in any direction on a graphic LCD.
LineTo	Draw a straight line in any direction on a graphic LCD, starting from the previous Line command's end position.
LoadBit	Set or Clear a bit of a port or variable, using a variable index.
LookDown	Search a constant lookdown table for a value.
LookDownL	Search constant or variable lookdown table for a value.
LookUp	Fetch a constant value from a lookup table.
LookUpL	Fetch a constant or variable value from lookup table.
Low	Make a pin or port low.
Lread	Read a value from an Ldata table.
Lread8, Lread16, Lread32	Read a single or multi-byte value from an Ldata table with more efficiency than Lread .
Mid\$	Extract characters from a String beginning at <i>n</i> characters from the left.
On Interrupt	Execute a subroutine using a Software interrupt (Legacy. Not Recommended).
On_Hardware_Interrupt	Point to the subroutine that an interrupt will jump too.
On_Low_Interrupt	Point to a subroutine for a Low Priority interrupt on an 18F device.
On Gosub	Call a Subroutine based on an Index value. For 18F devices only.
On Goto	Jump to an address in code memory based on an Index value. (Primarily for smaller devices)
On GotoL	Jump to an address in code memory based on an Index value. (Primarily for larger devices)
Output	Make a pin an output.
Oread	Receive data from a device using the Dallas 1-wire protocol.
Owrite	Send data to a device using the Dallas 1-wire protocol.
Org	Set Program Origin.
Pixel	Read a single pixel from a Graphic LCD.
Plot	Set a single pixel on a Graphic LCD.

Pot	Read a potentiometer on specified pin.
Print	Display characters on an LCD.
PulseIn	Measure the pulse width on a pin.
PulseOut	Generate a pulse to a pin.
Pwm	Output a pulse width modulated pulse train to pin.
Random	Generate a pseudo-random number.
RCin	Measure a pulse width on a pin.
Repeat...Until	Execute a block of instructions until a condition is true.
Resume	Re-enable software interrupts and return.
Return	Continue at the statement following the last Gosub .
Right\$	Extract <i>n</i> amount of characters from the right of a String.
Rsin	Asynchronous serial input from a fixed pin and baud rate.
Rout	Asynchronous serial output to a fixed pin and baud rate.
Seed	Seed the random number generator, to obtain a more random result.
Select..Case..EndSelect	Conditionally run blocks of code.
Serin	Receive asynchronous serial data (i.e. RS232 data).
Serout	Transmit asynchronous serial data (i.e. RS232 data).
Servo	Control a servo motor.
Set	Place a variable or bit in a high state.
Set_OSCCAL	Calibrate the internal oscillator found on some PICmicro™ devices.
SetBit	Set a bit of a port or variable, using a variable index.
Shin	Synchronous serial input.
Shout	Synchronous serial output.
Sleep	Power down the processor for a period of time.
Snooze	Power down the processor for short period of time.
Sound	Generate a tone or white-noise on a specified pin.
Sound2	Generate 2 tones from 2 separate pins.
Stop	Stop program execution.
Str	Load a Byte array with values.
Strn	Create a null terminated Byte array.
Str\$	Convert the contents of a variable to a null terminated String.
Swap	Exchange the values of two variables.
Symbol	Create an alias to a constant, port, pin, or register.
Toggle	Reverse the state of a port's bit.
ToLower	Convert the characters in a String to lower case.
ToUpper	Convert the characters in a String to upper case.
Toshiba_Command	Send a command to a Toshiba T6963 graphic LCD.
Toshiba_UDG	Create User Defined Graphics for Toshiba T6963 graphic LCD.
UnPlot	Clear a single pixel on a Graphic LCD.
USBinit	Initialise the USB interrupt on devices that contain a USB module.
USBin	Receive data via a USB endpoint on devices that contain a USB module.
USBout	Transmit data via a USB endpoint on devices that contain a USB module.
Val	Convert a null terminated String to an integer value.
VarPtr	Locate the address of a variable.
While...Wend	Execute statements while condition is true.
Xin	Receive data using the X10 protocol.
Xout	Transmit data using the X10 protocol.

Adin

Syntax

Variable = **Adin** *channel number*

Overview

Read the value from the on-board Analogue to Digital Converter.

Operators

Variable is a user defined variable.

Channel number can be a constant or a variable expression.

Example

```
' Read the value from channel 0 of the ADC and place in variable Var1.
  Declare Adin_Res = 10      ' 10-bit result required
  Declare Adin_Tad = FRC    ' RC oscillator chosen
  Declare Adin_Stime = 50   ' Allow 50us sample time

  Dim Var1 as Word
  TRISA = %00000001        ' Configure AN0 (PORTA.0) as an input
  ADCON1 = %10000000      ' Set analogue input on PORTA.0
  Var1 = Adin 0           ' Place the conversion into variable Var1
```

Adin Declares

There are three **Declare** directives for use with **Adin**. These are: -

Declare Adin_Res 8, 10, or 12.

Sets the number of bits in the result.

If this **Declare** is not used, then the default is the resolution of the PICmicro™ type used. For example, the 16F87X range will result in a resolution of 10-bits, along with the 18F devices, while the standard PICmicro™ types will produce an 8-bit result. Using the above **Declare** allows an 8-bit result to be obtained from the 10-bit PICmicro™ types, but not 10-bits from the 8-bit types.

Declare Adin_Tad 2_FOSC, 8_FOSC, 32_FOSC, 64_FOSC, or FRC.

Sets the ADC's clock source.

All compatible PICs have four options for the clock source used by the ADC. 2_FOSC, 8_FOSC, 32_FOSC, and 64_FOSC are ratios of the external oscillator, while FRC is the PICmicro's internal RC oscillator. Instead of using the predefined names for the clock source, values from 0 to 3 may be used. These reflect the settings of bits 0-1 in register ADCON0.

Care must be used when issuing this **Declare**, as the wrong type of clock source may result in poor resolution, or no conversion at all. If in doubt use FRC which will produce a slight reduction in resolution and conversion speed, but is guaranteed to work first time, every time. FRC is the default setting if the **Declare** is not issued in the BASIC listing.

Declare Adin_Stime 0 to 65535 microseconds (us).

Allows the internal capacitors to fully charge before a sample is taken. This may be a value from 0 to 65535 microseconds (us).

A value too small may result in a reduction of resolution. While too large a value will result in poor conversion speeds without any extra resolution being attained.

A typical value for **Adin_Stime** is 50 to 100. This allows adequate charge time without losing too much conversion speed. But experimentation will produce the right value for your particular requirement. The default value if the **Declare** is not used in the BASIC listing is 50.

Notes

Before the **Adin** command may be used, the appropriate **Tris** register must be manipulated to set the desired pin to an input. Also, the ADCON1 register must be set according to which pin is required as an analogue input, and in some cases, to configure the format of the conversion's result. See the numerous Microchip datasheets for more information on these registers and how to set them up correctly for the specific device used.

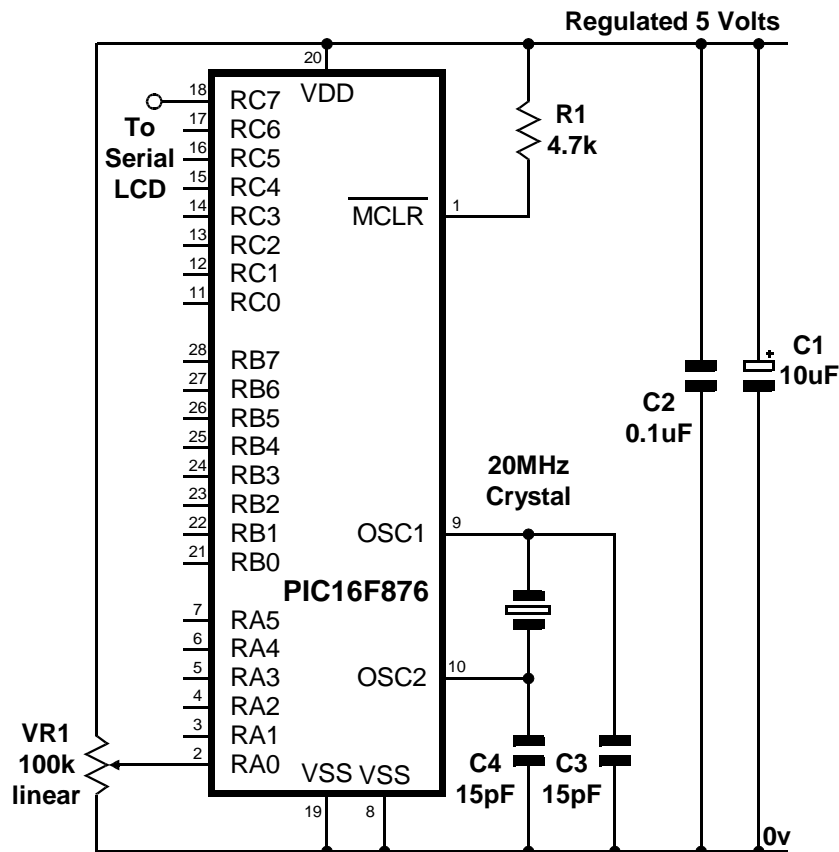
If multiple conversions are being implemented, then a small delay should be used after the **Adin** command. This allows the ADC's internal capacitors to discharge fully: -

Again:

```

Var1 = Adin 3 ' Place the conversion into variable Var1
DelayUs 1 ' Wait for 1us
Goto Again ' Read the ADC forever
    
```

The circuit below shows a typical setup for a simple ADC test.



See also : **Rcin, Pot.**

Asm..EndAsm

Syntax

Asm

assembler mnemonics

EndAsm

or

@ *assembler mnemonic*

Overview

Incorporate in-line assembler in the BASIC code. The mnemonics are passed directly to the assembler without the compiler interfering in any way. This allows a great deal of flexibility that cannot always be achieved using BASIC commands alone.

When the **Asm** directive is found within the BASIC program, the RAM banks are reset before the assembler code is operated upon. The same happens when the **EndAsm** directive is found, in that the RAM banks are reset upon leaving the assembly code. However, this may not always be required and can waste precious code memory. Placing a dash after **Asm** or **EndAsm** will remove the RAM reset mnemonics.

Asm-

EndAsm-

Only remove the RAM resets if you are confident enough to do so, as PICmicro™ devices have fragmented RAM.

The compiler also allows simple assembler mnemonics to be used within the BASIC program without wrapping them in **Asm-EndAsm**, however, the constants, labels, and variables used must be valid BASIC types.

Box

Syntax

Box *Set_Clear, Xpos Start, Ypos Start, Size*

Overview

Draw a square on a graphic LCD.

Operators

Set_Clear may be a constant or variable that determines if the square will set or clear the pixels. A value of 1 will set the pixels and draw a square, while a value of 0 will clear any pixels and erase a square .

Xpos Start may be a constant or variable that holds the X position for the centre of the square. Can be a value from 0 to 127.

Ypos Start may be a constant or variable that holds the Y position for the centre of the square. Can be a value from 0 to 63.

Size may be a constant or variable that holds the Size of the square (in pixels). Can be a value from 0 to 255.

Example

```
' Draw a square at position 63,32 with a size of 20 pixels  
' on a Samsung KS0108 graphic LCD
```

```
Include "Proton_G4.int"
```

```
Dim Xpos as Byte
```

```
Dim Ypos as Byte
```

```
Dim Size as Byte
```

```
Dim SetClr as Byte
```

```
DelayMs 100           ' Wait for the LCD to stabilise
```

```
Cls                   ' Clear the LCD
```

```
Xpos = 63
```

```
Ypos = 32
```

```
Size = 20
```

```
SetClr = 1
```

```
Box SetClr, Xpos, Ypos, Radius
```

```
Stop
```

Notes

Because of the aspect ratio of the pixels on the graphic LCD (approx 1.5 times higher than wide) the square will appear elongated.

See Also : **Circle, Line, LineTo, Plot, UnPlot.**

Branch

Syntax

Branch *Index*, [*Label1* {...*Labeln*}]

Overview

Cause the program to jump to different locations based on a variable index. On a PICmicro™ device with only one page of memory.

Operators

Index is a constant, variable, or expression, that specifies the address to branch to.

Label1,...**Labeln** are valid labels that specify where to branch to. A maximum of 255 labels may be placed between the square brackets, 256 if using an 18F device.

Example

```
Device = 16F84
Dim Index as Byte
Start:
  Index = 2           ' Assign Index a value of 2
  Branch Index,[Lab_0, Lab_1, Lab_2] ' Jump to Lab_2 because Index = 2
Lab_0:
  Index = 2           ' Index now equals 2
  Goto Start
Lab_1:
  Index = 0           ' Index now equals 0
  Goto Start
Lab_2:
  Index = 1           ' Index now equals 1
  Goto Start
```

The above example we first assign the index variable a value of 2, then we define our labels. Since the first position is considered 0 and the variable index equals 2 the **Branch** command will cause the program to jump to the third label in the brackets [Lab_2].

Notes

Branch operates the same as On x Goto. It's useful when you want to organise a structure such as: -

```
If Var1 = 0 Then Goto Lab_0   ' Var1 =0: go to label "Lab_0"
If Var1 = 1 Then Goto Lab_1   ' Var1 =1: go to label "Lab_1"
If Var1 = 2 Then Goto Lab_2   ' Var1 =2: go to label "Lab_2"
```

You can use **Branch** to organise this into a single statement: -

```
Branch Var1, [Lab_0, Lab_1, Lab_2]
```

This works exactly the same as the above **If...Then** example. If the value is not in range (in this case if Var1 is greater than 2), **Branch** does nothing. The program continues with the next instruction..

The **Branch** command is primarily for use with devices that have one page of memory (0-2047). If larger devices are used and you suspect that the branch label will be over a page boundary, use the **BranchL** command instead.

BranchL

Syntax

BranchL *Index*, [*Label1* {...*Labeln*}]

Overview

Cause the program to jump to different locations based on a variable index. On a PICmicro™ device with more than one page of memory.

Operators

Index is a constant, variable, or expression, that specifies the address to branch to.

Label1,...**Labeln** are valid labels that specify where to branch to. A maximum of 127 labels may be placed between the square brackets, 256 if using an 18F device.

Example

```
Device = 16F877
Dim Index as Byte
Start:
  Index = 2           ' Assign Index a value of 2
  ' Jump to label 2 (Label_2) because Index = 2
  BranchL Index,[Label_0, Label_1, Label_2]
Label_0:
  Index = 2           ' Index now equals 2
  Goto Start
Label_1:
  Index = 0           ' Index now equals 0
  Goto Start
Label_2:
  Index = 1           ' Index now equals 1
  Goto Start
```

The above example we first assign the index variable a value of 2, then we define our labels. Since the first position is considered 0 and the variable index equals 2 the **BranchL** command will cause the program to jump to the third label in the brackets [Label_2].

Notes

The **BranchL** command is mainly for use with PICmicro™ devices that have more than one page of memory (greater than 2048). It may also be used on any PICmicro™ device, but does produce code that is larger than **Branch**.

See also : **Branch**

Break

Syntax Break

Overview

Exit a **For...Next**, **While...Wend** or **Repeat...Until** loop prematurely.

Example 1

' Break out of a For-Next loop when the count reaches 10

```
Include "Proton_4.Inc"      ' Demo using Proton Dev board
Dim Var1 as Byte

DelayMs 100                ' Wait for the LCD to stabilise
Cls                        ' Clear the LCD
For Var1 = 0 to 39         ' Create a loop of 40 revolutions
    Print At 1,1,Dec Var1  ' Print the revolutions on the LCD
    If Var1 = 10 Then Break ' Break out of the loop when Var1 = 10
    DelayMs 200           ' Delay so we can see what's happening
Next                      ' Close the For-Next loop
Print At 2,1,"Exited At ", Dec Var1 ' Display value when loop was broke
Stop
```

Example 2

' Break out of a Repeat-Until loop when the count reaches 10

```
Include "Proton_4.Inc"      ' Demo using Proton Dev board
Dim Var1 as Byte

DelayMs 100                ' Wait for the LCD to stabilise
Cls                        ' Clear the LCD
Var1 = 0
Repeat                    ' Create a loop
    Print At 1,1,Dec Var1  ' Print the revolutions on the LCD
    If Var1 = 10 Then Break ' Break out of the loop when Var1 = 10
    DelayMs 200           ' Delay so we can see what's happening
    Inc Var1
Until Var1 > 39           ' Close the loop after 40 revolutions
Print At 2,1,"Exited At ", Dec Var1 ' Display value when loop was broke
Stop
```

Example 3

' Break out of a While-Wend loop when the count reaches 10

```
Include "Proton_4.Inc"      ' Demo using Proton Dev board
Dim Var1 as Byte
DelayMs 100                ' Wait for the LCD to stabilise
Cls                        ' Clear the LCD
Var1 = 0
While Var1 < 40            ' Create a loop of 40 revolutions
  Print At 1,1,Dec Var1    ' Print the revolutions on the LCD
  If Var1 = 10 Then Break ' Break out of the loop when Var1 = 10
  DelayMs 200             ' Delay so we can see what's happening
  Inc Var1
Wend                       ' Close the loop
Print At 2,1,"Exited At ", Dec Var1 ' Display value when loop was broke
Stop
```

Notes

The **Break** command is similar to a **Goto** but operates internally. When the **Break** command is encountered, the compiler will force a jump to the loop's internal exit label.

If the **Break** command is used outside of a **For...Next**, **Repeat...Until** or **While...Wend** loop, an error will be produced.

If the **Break** command is used within a **Select...EndSelect** construct while this is itself inside a loop, only the **Select...EndSelect** will be exited, not the loop.

See also : **Continue, For...Next, While...Wend, Repeat...Until.**

Bstart

Syntax

Bstart

Overview

Send a **Start** condition to the I²C bus.

Notes

Because of the subtleties involved in interfacing to some I²C devices, the compiler's standard **Busin**, and **Busout** commands were found lacking somewhat. Therefore, individual pieces of the I²C protocol may be used in association with the new structure of **Busin**, and **Busout**. See relevant sections for more information.

Example

```
' Interface to a 24LC32 serial eeprom
Device = 16F877
Dim Loop as Byte
Dim Array[10] as Byte
' Transmit bytes to the I2C bus
Bstart           ' Send a Start condition
Busout %10100000 ' Target an eeprom, and send a Write command
Busout 0         ' Send the High Byte of the address
Busout 0         ' Send the Low Byte of the address
For Loop = 48 to 57 ' Create a loop containing ASCII 0 to 9
  Busout Loop     ' Send the value of Loop to the eeprom
Next             ' Close the loop
Bstop           ' Send a Stop condition
DelayMs 10      ' Wait for the data to be entered into eeprom matrix
' Receive bytes from the I2C bus
Bstart           ' Send a Start condition
Busout %10100000 ' Target an eeprom, and send a Write command
Busout 0         ' Send the High Byte of the address
Busout 0         ' Send the Low Byte of the address
Brestart         ' Send a Restart condition
Busout %10100001 ' Target an eeprom, and send a Read command
For Loop = 0 to 9 ' Create a loop
  Array[Loop] = Busin ' Load an array with bytes received
  If Loop = 9 Then Bstop : Else : BusAck ' Ack or Stop ?
Next             ' Close the loop
Print At 1,1, Str Array ' Display the Array as a String
Stop
```

See also: **Bstop**, **Brestart**, **BusAck**, **Busin**, **Busout**, **HbStart**, **HbRestart**, **HbusAck**, **Hbusin**, **Hbusout**.

Bstop

Syntax
Bstop

Overview
Send a **Stop** condition to the I²C bus.

Brestart

Syntax
Brestart

Overview
Send a **Restart** condition to the I²C bus.

BusAck

Syntax
BusAck

Overview
Send an **Acknowledge** condition to the I²C bus.

BusNack

Syntax
BusNack

Overview
Send a **Not Acknowledge** condition to the I²C bus.

See also: **Bstop, Bstart, Brestart, Busin, Busout, HbStart, HbRestart, HbusAck, Hbusin, Hbusout.**

Busin

Syntax

Variable = **Busin** *Control*, { *Address* }

or

Variable = **Busin**

or

Busin *Control*, { *Address* }, [*Variable* {, *Variable*...}]

or

Busin *Variable*

Overview

Receives a value from the I²C bus, and places it into *variable*/s. If versions *two* or *four* (see above) are used, then No Acknowledge, or Stop is sent after the data. Versions *one* and *three* first send the *control* and optional *address*.

Operators

Variable is a user defined variable or constant.

Control may be a constant value or a **Byte** sized variable expression.

Address may be a constant value or a variable expression.

The four variations of the **Busin** command may be used in the same BASIC program. The *second* and *fourth* types are useful for simply receiving a single byte from the bus, and must be used in conjunction with one of the low level commands. i.e. **Bstart**, **Brestart**, **BusAck**, or **Bstop**. The *first*, and *third* types may be used to receive several values and designate each to a separate variable, or variable type.

The **Busin** command operates as an I²C master without using the microcontroller's MSSP peripheral, and may be used to interface with any device that complies with the 2-wire I²C protocol.

The most significant 7-bits of *control* byte contain the control code and the slave address of the device being interfaced with. Bit-0 is the flag that indicates whether a read or write command is being implemented.

For example, if we were interfacing to an external eeprom such as the 24LC32, the control code would be %10100001 or \$A1. The most significant 4-bits (1010) are the eeprom's unique slave address. Bits 1 to 3 reflect the three address pins of the eeprom. And bit-0 is set to signify that we wish to read from the eeprom. Note that this bit is automatically set by the **Busin** command, regardless of its initial setting.

Example

' *Receive a byte from the I2C bus and place it into variable Var1.*

```
Dim Var1 as Byte           ' We'll only read 8-bits
Dim Address as Word       ' 16-bit address required
Symbol Control %10100001  ' Target an eeprom
```

```
Address = 20 ' Read the value at address 20
Var1 = Busin Control, Address ' Read the byte from the eeprom
```

or

```
Busin Control, Address, [ Var1 ] ' Read the byte from the eeprom
```

Address, is an optional parameter that may be an 8-bit or 16-bit value. If a variable is used in this position, the size of *address* is dictated by the size of the variable used (**Byte**, **Word**, or **Dword**). In the case of the previous eeprom interfacing, the 24LC32 eeprom requires a 16-bit address. While the smaller types require an 8-bit address. Make sure you assign the right size address for the device interfaced with, or you may not achieve the results you intended.

The value received from the bus depends on the size of the variables used, except for variation three, which only receives a **Byte** (8-bits). For example: -

```
Dim Wrd as Word ' Declare a Word size variable
Wrd = Busin Control, Address
```

Will receive a 16-bit value from the bus. While: -

```
Dim Var1 as Byte ' Declare a Byte size variable
Var1 = Busin Control, Address
```

Will receive an 8-bit value from the bus.

Using the *third* variation of the **Busin** command allows differing variable assignments. For example: -

```
Dim Var1 as Byte
Dim Wrd as Word
Busin Control, Address, [Var1, Wrd]
```

Will receive two values from the bus, the first being an 8-bit value dictated by the size of variable *Var1* which has been declared as a byte. And a 16-bit value, this time dictated by the size of the variable *Wrd* which has been declared as a word. Of course, **Bit** type variables may also be used, but in most cases these are not of any practical use as they still take up a byte within the eeprom.

The *second* and *fourth* variations allow all the subtleties of the I²C protocol to be exploited, as each operation may be broken down into its constituent parts. It is advisable to refer to the datasheet of the device being interfaced to fully understand its requirements. See section on **Bstart**, **Brestart**, **BusAck**, or **Bstop**, for example code.

Declares

See **Busout** for declare explanations.

Notes

When the **Busout** command is used, the appropriate SDA and SCL Port and Pin are automatically setup as inputs, and outputs.

Because the I²C protocol calls for an *open-collector* interface, pull-up resistors are required on both the SDA and SCL lines. Values of 1KΩ to 4.7KΩ will suffice.

You may imagine that it's limiting having a fixed set of pins for the I²C interface, but you must remember that several different devices may be attached to a single bus, each having a unique slave address. Which means there is usually no need to use up more than two pins on the PICmicro™, in order to interface to many devices.

Str modifier with Busin

Using the **Str** modifier allows variations *three* and *four* of the **Busin** command to transfer the bytes received from the I²C bus directly into a byte array. If the amount of received characters is not enough to fill the entire array, then a formatter may be placed after the array's name, which will only receive characters until the specified length is reached. An example of each is shown below: -

```
Dim Array[10] as Byte      ' Define an array of 10 bytes
Dim Address as Byte       ' Create a word sized variable
Busin %10100000, Address, [Str Array] ' Load data into all the array
' Load data into only the first 5 elements of the array
Busin %10100000, Address, [Str Array\5]
Bstart                    ' Send a Start condition
Busout %10100000          ' Target an eeprom, and send a WRITE command
Busout 0                  ' Send the HighByte of the address
Busout 0                  ' Send the LowByte of the address
Brestart                  ' Send a Restart condition
Busout %10100001          ' Target an eeprom, and send a Read command
Busin Str Array           ' Load all the array with bytes received
Bstop                    ' Send a Stop condition
```

An alternative ending to the above example is: -

```
Busin Str Array\5        ' Load data into only the first 5 elements of the array
Bstop                    ' Send a Stop condition
```

See also : **BusAck, Bstart, Brestart, Bstop, Busout, HbStart, HbRestart, HbusAck, Hbusin, Hbusout.**

Busout

Syntax

Busout *Control*, { *Address* }, [*Variable* {, *Variable*...}]

or

Busout *Variable*

Overview

Transmit a value to the I²C bus, by first sending the *control* and optional *address* out of the clock pin (*SCL*), and data pin (*SDA*). Or alternatively, if only one operator is included after the **Busout** command, a single value will be transmitted, along with an Ack reception.

Operators

Variable is a user defined variable or constant.

Control may be a constant value or a **Byte** sized variable expression.

Address may be a constant, variable, or expression.

The **Busout** command operates as an I²C master using a bit-bashed (software only) method, and may be used to interface with any device that complies with the 2-wire I²C protocol.

The most significant 7-bits of *control* byte contain the control code and the slave address of the device being interfaced with. Bit-0 is the flag that indicates whether a read or write command is being implemented.

For example, if we were interfacing to an external eeprom such as the 24LC32, the control code would be %10100000 or \$A0. The most significant 4-bits (1010) are the eeprom's unique slave address. Bits 1 to 3 reflect the three address pins of the eeprom. And Bit-0 is clear to signify that we wish to write to the eeprom. Note that this bit is automatically cleared by the **Busout** command, regardless of its initial value.

Example

```
' Send a byte to the I2C bus.
Dim Var1 as Byte           ' We'll only read 8-bits
Dim Address as Word       ' 16-bit address required
Symbol Control = %10100000 ' Target an eeprom
Address = 20              ' Write to address 20
Var1 = 200               ' The value place into address 20
Busout Control, Address, [Var1] ' Send the byte to the eeprom
DelayMs 10                ' Allow time for allocation of byte
```

Address, is an optional parameter that may be an 8-bit or 16-bit value. If a variable is used in this position, the size of *address* is dictated by the size of the variable used (**Byte**, **Word** or **Dword**). In the case of the above eeprom interfacing, the 24LC32 eeprom requires a 16-bit address. While the smaller types require an 8-bit address. Make sure you assign the right size address for the device interfaced with, or you may not achieve the results you intended.

The value sent to the bus depends on the size of the variables used. For example: -

```
Dim Wrd as Word           ' Declare a Word size variable
Busout Control, Address, [Wrd]
```

Will send a 16-bit value to the bus. While: -

```
Dim Var1 as Byte         ' Declare a Byte size variable
Busout Control, Address, [Var1]
```

Will send an 8-bit value to the bus.

Using more than one variable within the brackets allows differing variable sizes to be sent. For example: -

```
Dim Var1 as Byte
Dim Wrd as Word
Busout Control, Address, [Var1, Wrd]
```

Will send two values to the bus, the first being an 8-bit value dictated by the size of variable Var1 which has been declared as a byte. And a 16-bit value, this time dictated by the size of the variable Wrd which has been declared as a word. Of course, **Bit** type variables may also be used, but in most cases these are not of any practical use as they still take up a byte within the eeprom.

A string of characters can also be transmitted, by enclosing them in quotes: -

```
Busout Control, Address, ["Hello World", Var1, Wrd]
```

Using the second variation of the **Busout** command, necessitates using the low level commands i.e. Bstart, Brestart, BusAck, or Bstop.

Using the **Busout** command with only one value after it, sends a byte of data to the I²C bus, and returns holding the Acknowledge reception. This acknowledge indicates whether the data has been received by the slave device.

The Ack reception is returned in the PICmicro's Carry flag, which is **STATUS.0**, and also System variable PP4.0. A value of zero indicates that the data was received correctly, while a one indicates that the data was not received, or that the slave device has sent a NAck return. You must read and understand the datasheet for the device being interfacing to, before the Ack return can be used successfully. An code snippet is shown below: -

```
' Transmit a byte to a 24LC32 serial eeprom
Dim PP4 as Byte System ' Bring the system variable into the BASIC program
Bstart                 ' Send a Start condition
Busout %10100000      ' Target an eeprom, and send a Write command
Busout 0               ' Send the High Byte of the address
Busout 0               ' Send the Low Byte of the address
Busout "A"            ' Send the value 65 to the bus
If PP4.0 = 1 Then Goto Not_Received ' Has Ack been received OK?
Bstop                  ' Send a Stop condition
DelayMs 10             ' Wait for the data to be entered into eeprom
```

Str modifier with Busout.

The **Str** modifier is used for transmitting a string of bytes from a byte array variable. A string is a set of bytes sized values that are arranged or accessed in a certain order. The values 1, 2, 3 would be stored in a string with the value 1 first, followed by 2 then followed by the value 3. A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string 1,2,3 would be stored in a byte array containing three bytes (elements).

Below is an example that sends four bytes from an array: -

```
Dim MyArray[10] as Byte      ' Create a 10-byte array.
MyArray [0] = "A"           ' Load the first 4 bytes of the array
MyArray [1] = "B"           ' With the data to send
MyArray [2] = "C"
MyArray [3] = "D"
Busout %10100000, Address, [Str MyArray\4] ' Send 4-byte string.
```

Note that we use the optional `\n` argument of **Str**. If we didn't specify this, the program would try to keep sending characters until all 10 bytes of the array were transmitted. Since we do not wish all 10 bytes to be transmitted, we chose to tell it explicitly to only send the first 4 bytes.

The above example may also be written as: -

```
Dim MyArray [10] as Byte    ' Create a 10-byte array.
Str MyArray = "ABCD"        ' Load the first 4 bytes of the array
Bstart                      ' Send a Start condition
Busout %10100000           ' Target an eeprom, and send a Write command
Busout 0                    ' Send the HighByte of the address
Busout 0                    ' Send the LowByte of the address
Busout Str MyArray\4       ' Send 4-byte string.
Bstop                      ' Send a Stop condition
```

The above example, has exactly the same function as the previous one. The only differences are that the string is now constructed using the **Str** as a command instead of a modifier, and the low-level Hbus commands have been used.

Declares

There are three **Declare** directives for use with **Busout**.

These are: -

Declare SDA_Pin Port . Pin

Declares the port and pin used for the data line (SDA). This may be any valid port on the PICmicro™. If this declare is not issued in the BASIC program, then the default Port and Pin is PORTA.0

Declare SCL_Pin Port . Pin

Declares the port and pin used for the clock line (SCL). This may be any valid port on the PICmicro™. If this declare is not issued in the BASIC program, then the default Port and Pin is PORTA.1

These declares, as is the case with all the Declares, may only be issued once in any single program, as they setup the I²C library code at design time.

Declare Slow_Bus On - Off or 1 - 0

Slows the bus speed when using an oscillator higher than 4MHz.

The standard speed for the I²C bus is 100KHz. Some devices use a higher bus speed of 400KHz. If you use an 8MHz or higher oscillator, the bus speed may exceed the devices specs, which will result in intermittent transactions, or in some cases, no transactions at all. Therefore, use this **Declare** if you are not sure of the device's spec. The datasheet for the device used will inform you of its bus speed.

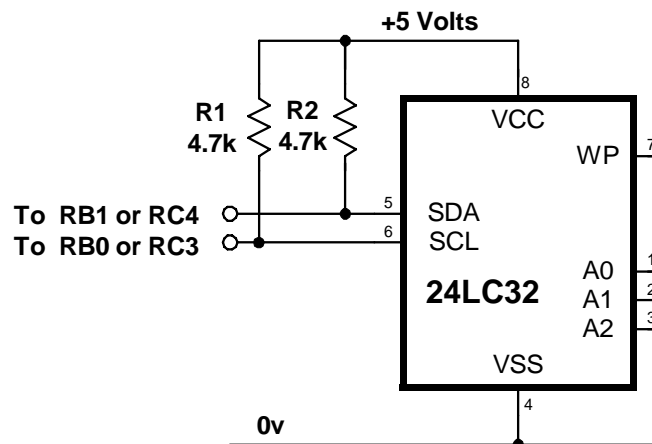
Notes

When the **Busout** command is used, the appropriate SDA and SCL Port and Pin are automatically setup as inputs, and outputs.

Because the I²C protocol calls for an *open-collector* interface, pull-up resistors are required on both the SDA and SCL lines. Values of 1KΩ to 4.7KΩ will suffice.

You may imagine that it's limiting having a fixed set of pins for the I²C interface, but you must remember that several different devices may be attached to a single bus, each having a unique slave address. Which means there is usually no need to use up more than two pins on the PICmicro™, in order to interface to many devices.

A typical use for the I²C commands is for interfacing with serial eeproms. Shown below is the connections to the I²C bus of a 24LC32 serial eeprom.



See also : **BusAck, Bstart, Brestart, Bstop, Busin, HbStart, HbRestart, HbusAck, Hbusin, Hbusout.**

Button

Syntax

Button *Pin, DownState, Delay, Rate, Workspace, TargetState, Label*

Overview

Debounce button input, perform auto-repeat, and branch to address if button is in target state. Button circuits may be active-low or active-high.

Operators

Pin is a Port.Bit, constant, or variable (0 - 15), that specifies the I/O pin to use. This pin will automatically be set to input.

DownState is a variable, constant, or expression (0 or 1) that specifies which logical state occurs when the button is pressed.

Delay is a variable, constant, or expression (0 - 255) that specifies how long the button must be pressed before auto-repeat starts. The delay is measured in cycles of the **Button** routine. Delay has two special settings: 0 and 255. If Delay is 0, **Button** performs no debounce or auto-repeat. If Delay is 255, **Button** performs debounce, but no auto-repeat.

Rate is a variable, constant, or expression (0 - 255) that specifies the number of cycles between auto-repeats. The rate is expressed in cycles of the **Button** routine.

Workspace is a byte variable used by **Button** for workspace. It must be cleared to 0 before being used by **Button** for the first time and should not be adjusted outside of the **Button** command.

TargetState is a variable, constant, or expression (0 or 1) that specifies which state the button should be in for a branch to occur. (0 = not pressed, 1 = pressed).

Label is a label that specifies where to branch if the button is in the target state.

Example

```
Dim BtnVar as Byte      ' Workspace for Button instruction.
Loop:                   ' Go to NoPress unless BtnVar = 0.
  Button 0, 0, 255, 250, BtnVar, 0, NoPress
  Print "*"
NoPress:
  Goto Loop
```

Notes

When a button is pressed, the contacts make or break a connection. A short (1 to 20ms) burst of noise occurs as the contacts scrape and bounce against each other. **Button**'s debounce feature prevents this noise from being interpreted as more than one switch action.

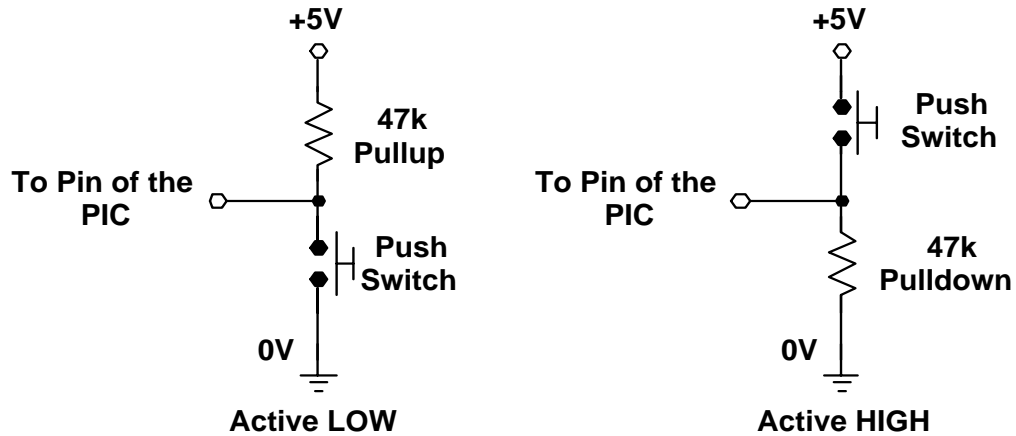
Button also reacts to a button press the way a computer keyboard does to a key press. When a key is pressed, a character immediately appears on the screen. If the key is held down, there's a delay, then a rapid stream of characters appears on the screen. **Button**'s auto-repeat function can be set up to work much the same way.

Button is designed for use inside a program loop. Each time through the loop, **Button** checks the state of the specified pin. When it first matches *DownState*, the switch is debounced. Then, as dictated by *TargetState*, it either branches to *address* (TargetState = 1) or doesn't (TargetState = 0).

If the switch stays in *DownState*, **Button** counts the number of program loops that execute. When this count equals *Delay*, **Button** once again triggers the action specified by *TargetState* and *address*. Thereafter, if the switch remains in *DownState*, **Button** waits *Rate* number of cycles between actions. The *Workspace* variable is used by **Button** to keep track of how many cycles have occurred since the *pin* switched to *TargetState* or since the last auto-repeat.

Button does not stop program execution. In order for its delay and auto repeat functions to work properly, **Button** must be executed from within a program loop.

Two suitable circuits for use with **Button** are shown below.



Call

Syntax

Call *Label*

Overview

Execute the assembly language subroutine named *label*.

Operators

Label must be a valid label name.

Example

```
' Call an assembler routine
Call Asm_Sub
```

Asm

```
Asm_Sub
{mnemonics}
Return
```

EndAsm

Notes

The **Gosub** command is usually used to execute a BASIC subroutine. However, if your subroutine happens to be written in assembler, the **Call** command should be used. The main difference between **Gosub** and **Call** is that when **Call** is used, the *label's* existence is not checked until assembly time. Using **Call**, a *label* in an assembly language section can be accessed that would otherwise be inaccessible to **Gosub**. This also means that any errors produced will be assembler types.

The **Call** command adds Page and Bank switching instructions prior to actually calling the subroutine, however, if **Call** is used in an all assembler environment, the extra mnemonics preceding the command can interfere with carefully sculptured code such as bit tests etc. By wrapping the subroutine's name in parenthesis, the Bank and Page instructions are suppressed, and the **Call** command becomes the **Call** mnemonic.

```
Call (Subroutine_Name)
```

Only use the mnemonic variation of **Call**, if you know that your destination is within the same Page as the section of code calling it. This is not an issue if using 18F devices, as they have a more linear memory organisation.

See also : **Gosub, Goto**

Cdata

Syntax

Cdata { *alphanumeric data* }

Overview

Place information directly into memory for access by **Cread** and **Cwrite**.

Operators

alphanumeric data can be any value, alphabetic character, or string enclosed in quotes (") or numeric data without quotes.

Example

```
Device = 16F877
Dim Var1 as Byte
Var1 = Cread 2000 ' Read the data from address 2000
Org 2000          ' Set the address of the Cdata command
Cdata 120         ' Place 120 at address 2000
```

In the above example, the data is located at address 2000 within the PICmicro™, then it's read using the **Cread** command.

Notes

Cdata is only available on the newer PICmicro™ types that have self-modifying features, such as the 16F87x range and the 18F devices, and offer an efficient use of precious code space.

The **Cread** and **Cwrite** commands can also use a label address as a location variable. For example: -

```
Device = 16F877          ' A device with code modifying features
Dim Dbyte as Byte
Dim Loop as Byte
For Loop = 0 to 9       ' Create a loop of 10
    Dbyte = Cread Address + Loop ' Read memory location Address + Loop
    Print Dbyte         ' Display the value read
Next
Stop
Address:
Cdata "HELLO WORLD" ' Create a string of text in flash memory
```

The program above reads and displays 10 values from the address located by the Label accompanying the **Cdata** command. Resulting in "HELLO WORL" being displayed.

Using the new in-line commands structure, the **Cread** and **Print** parts of the above program may be written as: -

```
' Read and display memory location Address + Loop
Print Cread Address + Loop
```

The **Cwrite** command uses the same technique for writing to memory: -

```
Device = 16F877           ' A device with code modifying features
Dim Dbyte as Byte
Dim Loop as Byte
' Write a string to code memory at location Address
Cwrite Address, ["HELLO WORLD"]
For Loop = 0 to 9         ' Create a loop of 10
    ' Read and display memory location Address + Loop
    Print Cread Address + Loop
Next
Stop
' Reserve 10 spaces in flash memory
Address:
Cdata 32, 32, 32, 32, 32, 32, 32, 32, 32, 32
```

Notice the string text now allowed in the **Cwrite** command. This allows the whole PICmicro™ to be used for data storage and retrieval if desired.

Important Note

Take care not to overwrite existing code when using the **Cwrite** command, and also remember that the all PICmicro™ devices have a finite amount of write cycles (approx 1000). A single program can easily exceed this limit, making that particular memory cell or cells inaccessible.

The configuration fuse setting **WRTE** must be enabled before **Cdata**, **Cread** and **Cwrite** may be used. This enables the self-modifying feature. If the **Config** directive is used, then the **WRTE_ON** fuse setting must be included in the list: -

```
Config WDT_ON, XT_OSC, WRTE_ON
```

Because the 14-bit core devices are only capable of holding 14 bits to a **Word**, values greater than 16383 (\$3FFF) cannot be stored.

Formatting a Cdata table with an 18F device.

Sometimes it is necessary to create a data table with a known format for its values. For example all values will occupy 4 bytes of data space even though the value itself would only occupy 1 or 2 bytes. Formatters are not supported with 14-bit core devices, because they can only hold a maximum value of \$3FFF (16383). i.e. 14-bits.

```
Cdata 100000, 10000, 1000, 100, 10, 1
```

The above line of code would produce an uneven code space usage, as each value requires a different amount of code space to hold the values. 100000 would require 4 bytes of code space, 10000 and 1000 would require 2 bytes, but 100, 10, and 1 would only require 1 byte.

Reading these values using **Cread** would cause problems because there is no way of knowing the amount of bytes to read in order to increment to the next valid value.

The answer is to use formatters to ensure that a value occupies a predetermined amount of bytes. These are: -

Byte
Word
Dword
Float

Placing one of these formatters before the value in question will force a given length.

```
Cdata Dword 100000, Dword 10000, Dword 1000, _  
      Dword 100, Dword 10, Dword 1
```

Byte will force the value to occupy one byte of code space, regardless of its value. Any values above 255 will be truncated to the least significant byte.

Word will force the value to occupy 2 bytes of code space, regardless of its value. Any values above 65535 will be truncated to the two least significant bytes. Any value below 255 will be padded to bring the memory count to 2 bytes.

Dword will force the value to occupy 4 bytes of code space, regardless of its value. Any value below 65535 will be padded to bring the memory count to 4 bytes. The line of code shown above uses the **Dword** formatter to ensure all the values in the **Cdata** table occupy 4 bytes of code space.

Float will force a value to its floating point equivalent, which always takes up 4 bytes of code space.

If all the values in an **Cdata** table are required to occupy the same amount of bytes, then a single formatter will ensure that this happens.

```
Cdata as Dword 100000, 10000, 1000, 100, 10, 1
```

The above line has the same effect as the formatter previous example using separate **Dword** formatters, in that all values will occupy 4 bytes, regardless of their value. All four formatters can be used with the **as** keyword.

The example below illustrates the formatters in use.

```
' Convert a Dword value into a string array  
' Using only BASIC commands  
' Similar principle to the Str$ command  
  
Include "Proton18_4.Inc"      ' Use an 18F device  
Dim P10 as Dword            ' Power of 10 variable  
Dim Cnt as Byte  
Dim J as Byte  
  
Dim Value as Dword          ' Value to convert  
Dim String1[11] as Byte    ' Holds the converted value
```

```
Dim Ptr as Byte      ' Pointer within the Byte array

DelayMs 100          ' Wait for the LCD to stabilise
Cls              ' Clear the LCD
Clear          ' Clear all RAM before we start
Value = 1234576    ' Value to convert
Gosub DwordToStr  ' Convert Value to string
Print Str String1  ' Display the result
Stop

' Convert a Dword value into a string array
' Value to convert is placed in 'Value'
' Byte array 'String1' is built up with the ASCII equivalent

DwordToStr:
Ptr = 0
J = 0
Repeat
P10 = Cread DwordTbl + (J * 4)
Cnt = 0

While Value >= P10
Value = Value - P10
Inc Cnt
Wend

If Cnt <> 0 Then
String1[Ptr] = Cnt + "0"
Inc Ptr
EndIf
Inc J
Until J > 8

String1[Ptr] = Value + "0"
Inc Ptr
String1[Ptr] = 0      ' Add the null to terminate the string
Return

' Cdata table is formatted for all 32-bit values.
' Which means each value will require 4 bytes of code space
Dword_TBL:
Cdata as Dword 1000000000, 100000000, 10000000, 1000000, 100000, _
               10000, 1000, 100, 10
```

Label names as pointers.

If a label's name is used in the list of values in a **Cdata** table, the labels address will be used. This is useful for accessing other tables of data using their address from a lookup table. See example below.

Note that this is not always permitted with standard 14-bit core devices, because they may not be able to hold the value in a 14-bit word.

8-bit Proton Compiler Development Suite.

```
' Display text from two Cdata tables
' Based on their address located in a separate table

Include "Proton18_4.Inc"      ' Use an 18F device

Dim Address as Word
Dim Loop as Word
Dim DataByte as Byte

DelayMs 100                  ' Wait for the LCD to stabilise
Cls                          ' Clear the LCD
Address = Cread AddrTable    ' Locate the address of the first string
While                        ' Create an infinite loop
  DataByte = Cread Address    ' Read each character from the Cdata string
  If DataByte = 0 Then Break ' Exit if null found
  Print DataByte             ' Display the character
  Inc Address                 ' Next character
Wend                          ' Close the loop

Cursor 2,1                   ' Point to line 2 of the LCD
Address = Cread AddrTable + 2 ' Locate the address of the second string
While 1 = 1                   ' Create an infinite loop
  DataByte = Cread Address    ' Read each character from the Cdata string
  If DataByte = 0 Then Break ' Exit if null found
  Print DataByte             ' Display the character
  Inc Address                 ' Next character
Wend                          ' Close the loop
Stop

AddrTable:                    ' Table of address's
  Cdata Word String1,Word String2
String1:
  Cdata "HELLO",0
String2:
  Cdata "WORLD",0
```

See also : **Config, Cread, Cread8, Cread16, Cread32, Cwrite, Dim, Ldata, Lread, Lread8, Lread16, Lread32.**

CF_Init

Syntax CF_Init

Overview

Initialise the lines used for Compact Flash access by **CF_Sector**, **CF_Read** and **CF_Write**.

Notes

CF_Init sets the pins used for the Compact Flash card to inputs and outputs accordingly. And must be issued before any Compact Flash commands are used in the program.

Essentially what the **CF_Init** command does can be shown by the BASIC code listed below: -

```
Low CF_DTPort      ' Set Data lines to output low
Low CF_ADPort      ' Set Address lines to output low
Output CF_WEPin    ' Set the CF WE pin to output
Low CF_CE1Pin      ' Set the CF CE1 pin to output low
Output CF_OEPin    ' Set the CF OE pin to output
Input CF_CD1Pin    ' Set the CF CD1 pin to input
Input CF_RDYPin    ' Set the CF RDY_BSY pin to input
High CF_RSTPin     ' Set the CF RESet pin to output high
DelayUs 1          ' Delay between toggles
Low CF_RSTPin      ' Set the CF RESet pin to output low
```

If the **CF_RSTPin Declare** is not issued in the BASIC program, then the **CF_RSTPin's** port.bit is not set up and no reset will occur through software. However, you must remember to tie the Compact Flash Reset pin to ground.

The same applies to the **CE1Pin**. If the **CF_CE1Pin Declare** is not issued in the BASIC program, then this pin is not manipulated in any way, and you must remember to tie the Compact Flash CE1 pin to ground

See Also: **CF_Sector** (for a suitable circuit), **CF_Read**, **CF_Write** (for declares).

CF_Sector

Syntax

CF_Sector *Sector Number, Operation, {Amount of Sectors}*

Overview

Setup the sector in the Compact Flash card that is to be written or read by the commands **CF_Read** and **CF_Write**.

Operators

Sector Number is the sector of interest in the Compact Flash card. This may be a constant value, variable, or expression. However, remember that there are potentially hundreds of thousands of sectors in a Compact Flash card so this variable will usually be a **Word** or **Dword** type.

Operation is the operation required by the Compact Flash card, this may either be the texts **WRITE** or **Read**. Or the values \$30 or \$20 which correspond to the texts accordingly.

Amount of Sectors is an optional parameter that informs the Compact Flash card as to how many sectors will be read or written in a single operation. This may be a constant value, variable, or expression. However, according to the Compact Flash data sheet, this may only be a value of 1 to 127, and is normally set to 1. The parameter is optional because it is usually only required once per **Read** or **Write** operation.

Example

```
' Write 20 sectors on a compact flash card, read back and display serially
Device = 18F452           ' We'll use an 18F device
Declare Xtal = 4

Declare Hserial_Baud = 9600           ' Set baud rate for USART serial coms
Declare Hserial_RCSTA = %10010000    ' Enable continuous receive
Declare Hserial_TXSTA = %00100100    ' Enable transmit and asynchronous mode
' CF Card Declarations
Declare CF_DTPort = PORTD             ' Assign the CF data port to PORTD
Declare CF_ADPort = PORTE             ' Assign the CF address port to PORTE
Declare CF_WEPin = PORTC.5           ' Assign the CF WE pin to PORTC.5
Declare CF_CE1Pin = PORTC.0          ' Assign the CF CE1 pin to PORTC.0
Declare CF_RDYPin = PORTC.4          ' Assign the CF RDY_BSY pin to PORTC.4
Declare CF_OEPin = PORTC.1           ' Assign the CF OE pin to PORTC.1
Declare CF_RSTPin = PORTC.3          ' Assign the CF RESet pin to PORTC.3
Declare CF_CD1Pin = PORTA.5          ' Assign the CF CD1 pin to PORTA.5
Declare CF_ADPort_Mask = False       ' No masking of address data required
Declare CF_Read_Write_Inline = False ' Use subroutines for CF_Read/CFWRITE

Symbol CF_CD1 = PORTA.5              ' Alias the CD1 pin to PORTA.5
' Variable Declarations

Dim Data_IO as Byte                  ' Bytes read/written to CF card
Dim BufferSize as Word                ' Internal counter of bytes in sector (i.e.512)
Dim SectorNumber as Dword            ' Sector of interest
' Main Program Starts Here
Delayms 100
ADCON1 = 7                           ' PORTA and PORTE to all digital mode
CF_Init                               ' Initialise the CF card's IO lines
While CF_CD1 = 1 : Wend               ' Is the Card inserted?
```

```
,
' Write 8-bit values from sector 0 to sector 20
,
WRITE_CF:
  Data_IO = 0           ' Clear the data to write to the card
  SectorNumber = 0     ' Start at sector 0

' Set up the CF card for Writing 1 sector at a time in LBA mode
CF_Sector SectorNumber,WRITE,1
Repeat                               ' Form a loop for the sectors
  BufferSize = 0
  Hserout ["WRITING SECTOR ",Dec SectorNumber,13]
  Repeat                               ' Form a loop for bytes in sector
    CF_Write [Data_IO]                 ' Write a byte to the CF card
    Inc BufferSize                       ' Move up a byte
    Inc Data_IO                         ' Increment the data to write
  Until BufferSize = 512                 ' Until all bytes are written
  Inc SectorNumber                     ' Move up to the next sector
  ' And Set up the CF card for Writing in LBA mode
  CF_Sector SectorNumber,Write
Until SectorNumber > 20               ' Until all sectors are written
,
' Read 8-bit values from sector 0 to sector 20
' And display serially In columns and rows format
,
Read_CF:
  SectorNumber = 0     ' Start at sector 0

' Set up the CF card for reading 1 sector at a time in LBA mode
CF_Sector SectorNumber,Read,1
Repeat                               ' Form a loop for the sectors
  BufferSize = 1
  Hserout ["SECTOR ",Dec SectorNumber,13]
  Repeat                               ' Form a loop for bytes in sector
    Data_IO = CF_Read                 ' Read a byte from the CF card
    Hserout [Hex2 Data_IO," "]         ' Display it in Hexadecimal
    If BufferSize // 32 = 0 Then Hserout [13] ' Check if row finished
    Inc BufferSize                       ' Move up a byte
  Until BufferSize > 512                 ' Until all bytes are read
  Hserout [Rep "-"\95,13]             ' Draw a line under each sector
  Inc SectorNumber                     ' Move up to the next sector
  ' And set up the CF card for reading in LBA mode
  CF_Sector SectorNumber,Read
Until SectorNumber > 20               ' Until all sectors are read
Stop
```

Example 2

```

' Display a summary of the Compact Flash
Device = 18F452           ' We'll use an 18F device
Declare Xtal = 4
Declare Hserial_Baud = 9600      ' Set baud rate for USART serial coms
Declare Hserial_RCSTA = %10010000 ' Enable continuous receive
Declare Hserial_TXSTA = %00100100 ' Enable transmit and asynchronous mode
' CF Card Declarations
Declare CF_DTPort = PORTD      ' Assign the CF data port to PORTD
Declare CF_ADPort = PORTE      ' Assign the CF address port to PORTE
Declare CF_WEPin = PORTC.5     ' Assign the CF WE pin to PORTC.5
Declare CF_CE1Pin = PORTC.0    ' Assign the CF CE1 pin to PORTC.0
Declare CF_RDYPin = PORTC.4    ' Assign the CF RDY_BSY pin to PORTC.4
Declare CF_OEPin = PORTC.1     ' Assign the CF OE pin to PORTC.1
Declare CF_RSTPin = PORTC.3    ' Assign the CF RESet pin to PORTC.3
Declare CF_CD1Pin = PORTA.5    ' Assign the CF CD1 pin to PORTA.5
Declare CF_ADPort_Mask = False  ' No masking of address data required
Declare CF_Read_Write_Inline = False ' Use subroutines for CF_Read/CFWRITE
Symbol CF_CD1 = PORTA.5       ' Alias the CD1 pin to PORTA.5
' Variable Declarations
Dim Data_IO as Word           ' Words read from CF card
Dim SER_Loop as Word          ' Internal counter of bytes
Dim SectorsPerCard as Dword   ' The amount of sectors in the CF card
Delaysms 100
ADCON1 = 7                    ' PORTA and PORTE to all digital mode
CF_Init                        ' Initialise the CF card's IO lines
While CF_CD1 = 1 : Wend       ' Is the Card inserted?
CF_Write 7,[$EC]              ' Write CF execute identify drive command
CF_Write $20,[]               ' Set address for Read SECTOR
Data_IO = CF_Read              ' Read from the CF card
Hserout ["General configuration = ",Hex4 Data_IO,13]
Data_IO = CF_Read              ' Read from the CF card
Hserout ["Default number of cylinders = ",Dec Data_IO,13]
Data_IO = CF_Read              ' Read from the CF card
Hserout ["Reserved = ",Dec Data_IO,13]
Data_IO = CF_Read              ' Read from the CF card
Hserout ["Default number of heads = ",Dec Data_IO,13]
Data_IO = CF_Read              ' Read from the CF card
Hserout ["Number of unformatted bytes per track = ",Dec Data_IO,13]
Data_IO = CF_Read              ' Read from the CF card
Hserout ["Number of unformatted bytes per sector = ",Dec Data_IO,13]
Data_IO = CF_Read              ' Read from the CF card
Hserout ["Default number of sectors per track = ",Dec Data_IO,13]
Data_IO = CF_Read              ' Read from the CF card
SectorsPerCard.HighWord = Data_IO
Data_IO = CF_Read              ' Read from the CF card
SectorsPerCard.LowWord = Data_IO
Hserout ["Number of sectors per card = ",Dec SectorsPerCard,13]
Data_IO = CF_Read              ' Read from the CF card
Hserout ["Vendor Unique = ",Dec Data_IO,13]
Hserout ["Serial number in ASCII (Right Justified) = "]
For SER_Loop = 0 to 19
    Data_IO.LowByte = CF_Read ' Read from the CF card
    Hserout [Data_IO.LowByte]
Next
Hserout [13]
Data_IO = CF_Read              ' Read from the CF card
Hserout ["Buffer type = ",Dec Data_IO,13]

```

8-bit Proton Compiler Development Suite.

```
Data_IO = CF_Read          ' Read from the CF card
Hserout ["Buffer size in 512 byte increments = ",Dec Data_IO,13]
Data_IO = CF_Read          ' Read from the CF card
Hserout ["# of ECC bytes passed on Read/Write Long Commands = ",_
        Dec Data_IO,13]

Stop
```

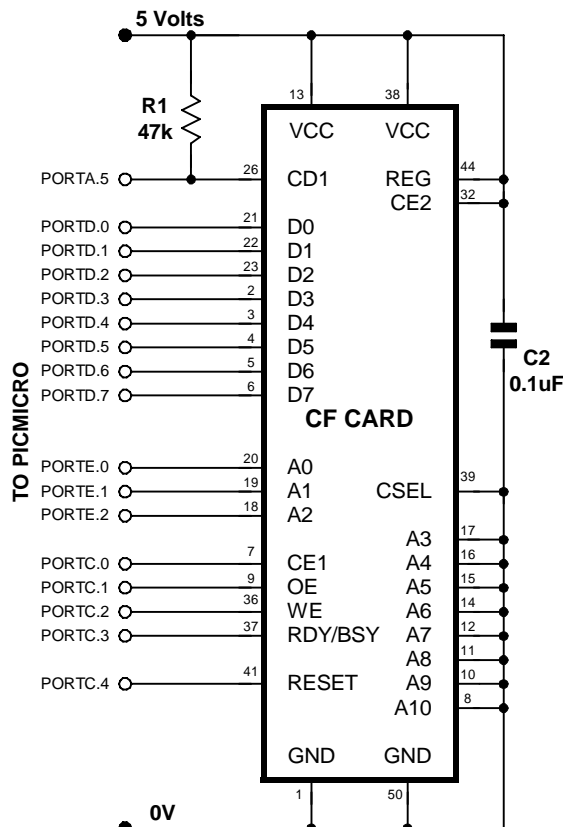
The above example will display on the serial terminal, some details concerning the Compact Flash card being interfaced. This is ideal for testing if the circuit is working, but is also useful for ascertaining how many sectors the Compact Flash card contains.

Notes

Accessing a compact flash card is not the same as accessing standard memory. In so much as a complete sector must be written. i.e. all 512 bytes in a single operation. Reading from a compact flash card is more conventional in that once the sector is chosen using the **CF_Sector** command, any of the 512 bytes may be read from that sector.

The compiler's Compact Flash access commands operate in what is called LBA (Logical Block Address) mode. Which means that it is accessed sector by sector instead of the more involved Cylinder/Head/Sector mode. LBA mode makes accessing Compact Flash easier and more intuitive. However, it is important to read and understand the CF+ and Compact Flash specifications document which can be obtained via the internet at www.compactflash.org.

A typical circuit for interfacing a Compact Flash card is shown below: -



The circuit shown overleaf can be used with the code examples listed earlier.

The Reset and CE1 lines are not essential to the operation of interfacing. The Reset line and the CE1 line must be connected to ground. However, the CE1 line is useful if multiplexing is used as the Compact Flash card will ignore all commands if the CE1 line is set high. And the Reset line is useful for a clean start up of the Compact Flash card.

The CF commands were written and tested only on the more modern “higher speed” compact flash cards. These operate at up to 40 times faster than conventional Compact Flash and also, more importantly, operate from a 3.3 Volt and 5 Volt power source. However, the low level routines used by the commands, when not in inline mode, are contained in a separate Inc file located inside the compiler’s **Includes** folder. The file is named **cf_cms.inc**, and can be altered if slower access is required. It is simply a matter of adding more **Nop** mnemonics inside the **CF@WR** and **CF@RD** subroutines.

See Also: **CF_Init, CF_Read, CF_Write (for declares).**

CF_Read

Syntax

Variable = **CF_Read**

Overview

Read data from a Compact Flash card.

Operators

Variable is a variable that will be loaded with data read from the Compact Flash card.

Example

```
' Read 16-bit values from 20 sectors in a compact flash card display
Device = 16F877           ' We'll use a 14-bit core device
Declare Xtal = 4

Declare Hserial_Baud = 9600           ' Set baud rate for USART serial coms
Declare Hserial_RCSTA = %10010000    ' Enable continuous receive
Declare Hserial_TXSTA = %00100100    ' Enable transmit and asynchronous mode
-----

' CF Card Declarations
Declare CF_DTPort = PORTD             ' Assign the CF data port to PORTD
Declare CF_ADPort = PORTE            ' Assign the CF address port to PORTE
Declare CF_WEPin = PORTC.5           ' Assign the CF WE pin to PORTC.5
Declare CF_CE1Pin = PORTC.0          ' Assign the CF CE1 pin to PORTC.0
Declare CF_RDYPin = PORTC.4          ' Assign the CF RDY_BSY pin to PORTC.4
Declare CF_OEPin = PORTC.1           ' Assign the CF OE pin to PORTC.1
Declare CF_RSTPin = PORTC.3          ' Assign the CF RESet pin to PORTC.3
Declare CF_CD1Pin = PORTA.5          ' Assign the CF CD1 pin to PORTA.5
Declare CF_ADPort_Mask = False       ' No masking of address data required
Declare CF_Read_Write_Inline = False ' Use subroutines for CF_Read/CFWRITE
-----

Symbol CF_CD1 = PORTA.5              ' Alias the CD1 pin to PORTA.5
-----

' Variable Declarations

Dim Data_IO as Word                  ' Words read from CF card
Dim BufferSize as Word                ' Internal counter of bytes in sector (i.e.512)
Dim SectorNumber as Dword            ' Sector of interest
-----

' Main Program Starts Here
Delaysms 100
ADCON1 = 7
CF_Init                               ' Initialise the CF card's IO lines
While CF_CD1 = 1 : Wend               ' Is the Card inserted?
-----

' Read 8-bit values from sector 0 to sector 20
' And display serially In columns and rows format
Read_CF:
SectorNumber = 0                      ' Start at sector 0
' Set up the CF card for reading 1 sector at a time in LBA mode
CF_Sector SectorNumber,Read,1
Repeat                                 ' Form a loop for the sectors
    BufferSize = 1
    Hserout ["SECTOR ",Dec SectorNumber,13]
    Repeat                               ' Form a loop for words in sector
        Data_IO = CF_Read                ' Read a Word from the CF card
```

```
Hserout [Hex4 Data_IO," "]           ' Display it in Hexadecimal
If BufferSize // 32 = 0 Then Hserout [13] ' Check if row finished
Inc BufferSize                       ' Move up a word
Until BufferSize > 256                ' Until all words are read
Hserout [Rep "-"\95,13]              ' Draw a line under each sector
Inc SectorNumber                     ' Move up to the next sector
' And set up the CF card for reading in LBA mode
CF_Sector SectorNumber,Read
Until SectorNumber > 20              ' Until all sectors are read
Stop
```

Notes

The amount of bytes read from the Compact Card depends on the variable type used as the assignment. i.e. the variable before the equals operator: -

A **Bit** type variable will read 1 byte from the Compact Flash card.

A **Byte** type variable will also read 1 byte from the Compact Flash card.

A **Word** type variable will read 2 bytes from the Compact Flash card Least Significant Byte First (LSB).

A **Dword** type variable will read 4 bytes from the Compact Flash card Least Significant Byte First (LSB).

A **Float** type variable will also read 4 bytes from the Compact Flash card in the correct format for a floating point variable.

Accessing Compact Flash memory is not the same as conventional memory. There is no mechanism for choosing the address of the data in question. You can only choose the sector then sequentially read the data from the card. In essence, the sector is the equivalent of the address in a conventional piece of memory, but instead of containing 1 byte of data, it contains 512 bytes.

Once the sector is chosen using the **CF_Sector** command, any amount of the 512 bytes available can be read from the card. Once a read has been accomplished, the Compact Flash card automatically increments to the next byte in the sector ready for another read. So that a simple loop as shown below will read all the bytes in a sector: -

```
BufferSize = 0
Repeat                                           ' Form a loop for bytes in sector
  Data_IO = CF_Read                             ' Read a Byte from the CF card
  Inc BufferSize                                 ' Increment the byte counter
Until BufferSize = 512                          ' Until all Bytes are read
```

In order to extract a specific piece of data from a sector, a similar loop can be used, but with a condition attached that will drop out at the correct position: -

```
BufferSize = 0
While                                     ' Form an infinite loop
  Data_IO = CF_Read                       ' Read a Byte from the CF card
  If BufferSize = 20 Then Break           ' Exit when correct position reached
  Inc BufferSize                           ' Increment the byte counter
Wend                                       ' Close the loop
```

The snippet above will exit the loop when the 20th byte has been read from the card.

Of course Arrays can also be loaded from a Compact Flash card in a similar way, but remember, the maximum size of an array in Proton BASIC is 256 elements. The snippets below show two possible methods of loading an array with the data read from a Compact Flash card.

```
Dim Ar1[256] as Byte                       ' Create a 256 element array
Dim BufferSize as Word                     ' Internal counter of bytes in sector
BufferSize = 0
Repeat                                     ' Form a loop for bytes in sector
  Ar1[BufferSize] = CF_Read               ' Read a Byte from the CF card
  Inc BufferSize                           ' Increment the byte counter
Until BufferSize = 256                    ' Until all Bytes are read
```

Large arrays such as the one above are best suited to the 18F devices. Not only because they generally have more RAM, but because their RAM is accessed more linearly and there are no Bank boundaries when using arrays. Also, by accessing some low level registers in an 18F device it is possible to efficiently place all 512 bytes from a sector into 2 arrays:

```
Device = 18F452                            ' Choose an 18F device
Dim Ar1[256] as Byte                       ' Create a 256 element array
Dim Ar2[256] as Byte                       ' Create another 256 element array
Dim BufferSize as Word                     ' Internal counter of bytes in sector
Dim FSR0 as FSR0L.Word                    ' Combine FSR0L/H as a 16-bit register
BufferSize = 0
FSR0 = AddressOf(Ar1)                     ' Get the address of Ar1 into FSR0L/H
Repeat                                     ' Form a loop for bytes in sector
  POSTINC0 = CF_Read                       ' Read a Byte from the CF card and place
                                           ' directly into memory, then increment to
                                           ' the next address in PIC RAM
                                           '
  Inc BufferSize                             ' Increment the byte counter
Until BufferSize = 512                     ' Until all Bytes are read
```

When the above loop is finished, arrays Ar1 and Ar2 will hold the data read from the Compact Flash card's sector. Of course you will need to pad out the snippets with the appropriate declares and the **CF_Sector** command.

See Also: **CF_Init**, **CF_Sector** (for a suitable circuit), **CF_Write** (for declares).

CF_Write

Syntax

CF_Write {Address Data}, [Variable {Variable {, Variable etc}}]

Overview

Write data to a Compact Flash card.

Operators

Address Data is an optional value that is placed on the Compact Flash card's Address lines. This is not always required when writing to a card.

Variable is the variable or constant value that will be written to the Compact Flash card. More than one variable can be placed between the square braces if more than one write is required in a single operation.

The *variable* part of the **CF_Write** command is also optional, as some configurations only require the card's address lines to be loaded. In this case, use the syntax: -

CF_Write Address Data, []

See example 2 in the CF_Sector section for an example of its use.

Example

```
' Write 20 sectors on a compact flash card
Device = 18F452           ' We'll use an 18F device
Declare Xtal = 4
Declare Hserial_Baud = 9600 ' Set baud rate for USART serial coms
Declare Hserial_RCSTA = %10010000 ' Enable continuous receive
Declare Hserial_TXSTA = %00100100 ' Enable transmit and asynchronous mode
-----
' CF Card Declarations
Declare CF_DTPort = PORTD ' Assign the CF data port to PORTD
Declare CF_ADPort = PORTE ' Assign the CF address port to PORTE
Declare CF_WEPin = PORTC.5 ' Assign the CF WE pin to PORTC.5
Declare CF_CE1Pin = PORTC.0 ' Assign the CF CE1 pin to PORTC.0
Declare CF_RDYPin = PORTC.4 ' Assign the CF RDY_BSY pin to PORTC.4
Declare CF_OEPin = PORTC.1 ' Assign the CF OE pin to PORTC.1
Declare CF_RSTPin = PORTC.3 ' Assign the CF RESet pin to PORTC.3
Declare CF_CD1Pin = PORTA.5 ' Assign the CF CD1 pin to PORTA.5
Declare CF_ADPort_Mask = False ' No masking of address data required
Declare CF_Read_Write_Inline = False ' Use subroutines for CF_Read/CFWRITE

Symbol CF_CD1 = PORTA.5 ' Alias the CD1 pin to PORTA.5
-----
' Variable Declarations

Dim Data_IO as Byte ' Bytes written to CF card
Dim BufferSize as Word ' Internal counter of bytes in sector (i.e.512)
Dim SectorNumber as Dword ' Sector of interest
-----
' Main Program Starts Here
Delayms 100
ADCON1 = 7 ' PORTA and PORTE to all digital mode
```

```
CF_Init           ' Initialise the CF card's IO lines
While CF_CD1 = 1 : Wend      ' Is the Card inserted?
'-----
' WRITE 8-bit values from sector 0 to sector 20
WRITE_CF:
Data_IO = 0           ' Clear the data to write to the card
SectorNumber = 0      ' Start at sector 0
' Set up the CF card for Writing 1 sector at a time in LBA mode
CF_Sector SectorNumber,Write,1
Repeat              ' Form a loop for the sectors
  BufferSize = 0
  Hserout ["WRITING SECTOR ",Dec SectorNumber,13]
  Repeat          ' Form a loop for bytes in sector
    CF_Write [Data_IO]      ' Write a byte to the CF card
    Inc BufferSize          ' Move up a byte
    Inc Data_IO            ' Increment the data to write
  Until BufferSize = 512    ' Until all bytes are written
  Inc SectorNumber        ' Move up to the next sector
  ' And Set up the CF card for Writing in LBA mode
  CF_Sector SectorNumber,Write
Until SectorNumber > 20    ' Until all sectors are written
Stop
```

Notes

The amount of bytes written to the Compact Card depends on the variable type used between the square braces: -

A **Bit** type variable will write 1 byte to the CF card.

A **Byte** type variable will also write 1 byte to the CF card.

A **Word** type variable will write 2 bytes to the CF card Least Significant Byte First (LSB).

A **Dword** type variable will write 4 bytes to the CF card Least Significant Byte First (LSB).

A **Float** type variable will also write 4 bytes to the CF card in the correct format of a floating point variable.

Accessing Compact Flash memory is not the same as conventional memory. There is no mechanism for choosing the address of the data in question. You can only choose the sector then sequentially write the data to the card. In essence, the sector is the equivalent of the address in a conventional piece of memory, but instead of containing 1 byte of data, it contains 512 bytes.

Once the sector is chosen using the **CF_Sector** command and a write operation is started, all 512 bytes contained in the sector must be written before they are transferred to the card's flash memory.

Once a single write has been accomplished, the Compact Flash card automatically increments to the next byte in the sector ready for another write. So that a simple loop as shown below will write all the bytes in a sector: -

```
BufferSize = 0
Repeat      ' Form a loop for bytes in sector
  CF_Write [Data_IO] ' Write a Byte to the CF card
  Inc BufferSize     ' Increment the byte counter
Until BufferSize = 512 ' Until all Bytes are written
```

Compact Flash Interface Declares

There are several declares that need to be manipulated when interfacing to a Compact Flash card. There are the obvious port pins, but there are also some declares that optimise or speed up access to the card.

Declare **CF_DTPort** Port

This declare assigns the Compact Flash card's data lines. The data line consists of 8-bits so it is only suitable for ports that contain 8-bits such as PORTB, PORTC, PORTD etc.

Declare **LCD_ADPort** Port

This declare assigns the Compact Flash card's address lines. The address line consists of 3-bits, but A0 of the compact flash card must be attached to bit-0 of whatever port is used. For example, if the Compact Flash card's address lines were attached to PORTA of the PICmicro™, then A0 of the CF card must attach to PORTA.0, A1 or the CF card must attach to PORTA.1, and A2 of the CF card must attach to PORTA.2.

The CF access commands will mask the data before transferring it to the particular port that is being used so that the rest of it's pins are not effected. PORTE is perfect for the address lines as it contains only 3 pins on a 40-pin device, and the compiler can make full use of this by using the **CF_ADPort_Mask** declare.

Declare **CF_ADPort_Mask = On** or **Off**, or **True** or **False**, or 1, 0

Both the **CF_Write** and **CF_Sector** commands write to the Compact Flash card's address lines. However, these only contain 3-bits, so the commands need to ensure that the other bits of the PICmicro's Port are not effected. This is accomplished by masking the unwanted data before transferring it to the address lines. This takes a little extra code space, and thus a little extra time to accomplish. However, there are occasions when the condition of the other bits on the Port are not important, or when a Port is used that only has 3-bits to it. i.e. PORTE with a 40-pin device. Issuing the **CF_ADPort_Mask** declare and setting it False, will remove the masking mnemonics, thus reducing code used and time taken.

Declare **CF_RDYPin** Port . Pin

Assigns the Compact Flash card's RDY/BSY line.

Declare **CF_OEPin** Port . Pin

Assigns the Compact Flash card's OE line.

Declare **CF_WEPin** Port . Pin

Assigns the Compact Flash card's WE line.

Declare CF_CD1Pin Port . Pin

Assigns the Compact Flash card's CD1 line. The CD1 line is not actually used by any of the commands, but is set to input if the declare is issued in the BASIC program. The CD1 line is used to indicate whether the card is inserted into its socket.

Declare CF_RSTPin Port . Pin

Assigns the Compact Flash card's RESet line. The RESet line is not essential for interfacing to a Compact Flash card, but is useful if a clean power up is required. If the declare is not issued in the BASIC program, all reference to it is removed from the **CF_Init** command. If the RESet line is not used for the card, ensure that it is tied to ground.

Declare CF_CE1Pin Port . Pin

Assigns the Compact Flash card's CE1 line. As with the RESet line, the CE1 line is not essential for interfacing to a Compact Flash card, but is useful when multiplexing pins, as the card will ignore all commands when the CE1 line is set high. If the declare is not issued in the BASIC program, all reference to it is removed from the **CF_Init** command. If the CE1 line is not used for the card, ensure that it is tied to ground.

Declare CF_Read_Write_Inline = On or Off, or True or False, or 1, 0

Sometimes, speed is of the essence when accessing a Compact Flash card, especially when interfacing to the new breed of card which is 40 times faster than the normal type. Because of this, the compiler has the ability to create the code used for the **CF_Write** and **CF_Read** commands inline, which means it does not call its library subroutines, and can tailor itself when reading or writing **Word**, **Dword**, or **Float** variables. However, this comes at a price of code memory, as each command is stretched out for speed, not optimisation. It also means that the inline type of commands are really only suitable for the higher speed Compact Flash cards.

If the declare is not used in the BASIC program, the default is not to use inline commands.

Circle

Syntax

Circle *Set_Clear, Xpos, Ypos, Radius*

Overview

Draw a circle on a graphic LCD.

Operators

Set_Clear may be a constant or variable that determines if the circle will set or clear the pixels. A value of 1 will set the pixels and draw a circle, while a value of 0 will clear any pixels and erase a circle.

Xpos may be a constant or variable that holds the X position for the centre of the circle. Can be a value from 0 to the X resolution of the display.

Ypos may be a constant or variable that holds the Y position for the centre of the circle. Can be a value from 0 to the Y resolution of the display.

Radius may be a constant or variable that holds the Radius of the circle. Can be a value from 0 to 255.

Example

```
' Draw circle at pos 63,32 with radius of 20 pixels on a Samsung KS0108 LCD
```

```
Include "Proton_G4.int" ' Use a Samsung KS0108 LCD
```

```
Dim Xpos as Byte
```

```
Dim Ypos as Byte
```

```
Dim Radius as Byte
```

```
Dim SetClr as Byte
```

```
DelayMs 100 ' Wait for the LCD to stabilise
```

```
Cls ' Clear the LCD
```

```
Xpos = 63
```

```
Ypos = 32
```

```
Radius = 20
```

```
SetClr = 1
```

```
Circle SetClr, Xpos, Ypos, Radius
```

```
Stop
```

Notes

Because of the aspect ratio of the pixels on the graphic LCD (approx 1.5 times higher than wide) the circle will appear elongated.

See Also : [Box](#), [Line](#), [Pixel](#), [Plot](#), [UnPlot](#).

Clear

Syntax

Clear *Variable* or *Variable.Bit*

Clear

Overview

Place a variable or bit in a low state. For a variable, this means loading it with 0. For a bit this means setting it to 0.

Clear has another purpose. If no variable is present after the command, all user RAM within the device is cleared.

Operators

Variable can be any variable or register.

Variable.Bit can be any variable and bit combination.

Example

```
Clear           ' Clear all RAM area
Clear Var1.3    ' Clear bit 3 of Var1
Clear Var1      ' Load Var1 with the value of 0
Clear STATUS.0 ' Clear the carry flag high
Clear Array     ' Clear all of an Array variable. i.e. reset to zero's
Clear String1   ' Clear all of a String variable. i.e. reset to zero's
```

Notes

There is a major difference between the **Clear** and **Low** command. **Clear** does not alter the TRIS register if a Port is targeted.

See Also : **Set, Low, High**

ClearBit

Syntax

ClearBit *Variable, Index*

Overview

Clear a bit of a variable or register using a variable index to the bit of interest.

Operators

Variable is a user defined variable.

Index is a constant, variable, or expression that points to the bit within *Variable* that requires clearing.

Example

```
' Clear then Set each bit of variable ExVar
Device = 16F877
Declare Xtal = 4
Dim ExVar as Byte
Dim Index as Byte
Cls
ExVar = %11111111

While                                     ' Create an infinite loop
  For Index = 0 to 7                       ' Create a loop for 8 bits
    ClearBit ExVar, Index                  ' Clear each bit of ExVar
    Print At 1,1,Bin8 ExVar                ' Display the binary result
    DelayMs 100                            ' Slow things down to see what's happening
  Next                                     ' Close the loop
  For Index = 7 to 0 Step -1              ' Create a loop for 8 bits
    SetBit ExVar, Index                    ' Set each bit of ExVar
    Print At 1,1,Bin8 ExVar                ' Display the binary result
    DelayMs 100                            ' Slow things down to see what's happening
  Next                                     ' Close the loop
Wend                                       ' Do it forever
```

Notes

There are many ways to clear a bit within a variable, however, each method requires a certain amount of manipulation, either with rotates, or alternatively, the use of indirect addressing using the **FSR**, and **INDF** registers. Each method has its merits, but requires a certain amount of knowledge to accomplish the task correctly. The **ClearBit** command makes this task extremely simple using a register rotate method, however, this is not necessarily the quickest method, or the smallest, but it is the easiest. For speed and size optimisation, there is no shortcut to experience.

To Clear a known constant bit of a variable or register, then access the bit directly using Port.n.

```
PORTA.1 = 0
or
Var1.4 = 0
```

If a Port is targeted by **ClearBit**, the Tris register is **not** affected.

See also : **GetBit, LoadBit, SetBit.**

Cls

Syntax Cls

Or if using a Toshiba T6963 graphic LCD

Cls Text Cls Graphic

Overview

Clears the alphanumeric or graphic LCD and places the cursor at the home position i.e. line 1, position 1 (line 0, position 0 for graphic LCDs).

Toshiba graphic LCDs based upon the T6963 chipset have separate RAM for text and graphics. Issuing the word **Text** after the **Cls** command will only clear the TEXT RAM, while issuing the word **Graphic** after the **Cls** command will only clear the Graphic RAM. Issuing the **Cls** command on its own will clear both areas of RAM.

Example 1

```
' Clear an alphanumeric or Samsung KS0108 graphic LCD
Cls          ' Clear the LCD
Print "HELLO" ' Display the word "HELLO" on the LCD
Cursor 2, 1  ' Move the cursor to line 2, position 1
Print "WORLD" ' Display the word "WORLD" on the LCD
Stop
```

In the above example, 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.

Example 2

```
' Clear a Toshiba T6963 graphic LCD.
Cls          ' Clear all RAM within the LCD
Print "HELLO" ' Display the word "HELLO" on the LCD
Line 1,0,0,63,63 ' Draw a line on the LCD
DelayMs 1000  ' Wait for 1 second
Cls Text     ' Clear only the text RAM, leaving the line displayed
DelayMs 1000 ' Wait for 1 second
Cls Graphic  ' Now clear the line from the display
Stop
```

Notes

The **Cls** command will also initialise any of the above LCDs. (set the ports to inputs/outputs etc), however, this is most important to Toshiba graphic LCDs, and the **Cls** command should always be placed at the head of the BASIC program, prior to issuing any command that interfaces with the LCD. i.e. **Print**, **Plot** etc.

See also : [Cursor](#), [Print](#), [Toshiba_Command](#).

Config

Syntax

Config { *configuration fuse settings* }

Overview

Enable or Disable particular fuse settings for the PICmicro™.

Operators

configuration fuse settings vary from device to device, however, certain settings are standard to most PICmicro™ types. Refer to the microcontroller's datasheet for details.

Example

' *Disable Watchdog timer and specify an HS_OSC etc, on a PIC16F877 device*

```
Config HS_OSC, WDT_OFF, PWRTE_ON, BODEN_OFF, LVP_OFF, _  
WRTE_ON, CP_OFF, DEBUG_OFF
```

18F Fuse Setting.

Because of the complexity that 18F devices require for adjusting their many fuses, the **Config** directive is not suitable. Instead a more intuitive approach is adopted using the **Config_Start** and **Config_End** directives: -

Config_Start

```
OSC = HS           ' Oscillator Selection HS  
OSCS = Off        ' Osc. Switch Enable Disabled  
PWRT = On         ' Power-up Timer Enabled  
BOR = Off         ' Brown-out Reset Disabled  
BORV = 25         ' Brown-out Voltage 2.5V  
WDT = Off        ' Watchdog Timer Disabled  
WDTPS = 128      ' Watchdog Postscaler 1:128  
CCP2MUX = On     ' CCP2 MUX Enable (RC1)  
STVR = Off       ' Stack Overflow Reset Disabled  
LVP = Off        ' Low Voltage ICSP Disabled  
DEBUG = Off      ' Background Debugger Enable Disabled  
CP0 = Off        ' Code Protection Block 0 Disabled  
CP1 = Off        ' Code Protection Block 1 Disabled  
CP2 = Off        ' Code Protection Block 2 Disabled  
CP3 = Off        ' Code Protection Block 3 Disabled  
CPB = Off        ' Boot Block Code Protection Disabled  
CPD = Off        ' Data EEPROM Code Protection Disabled  
WRT0 = Off       ' Write Protection Block 0 Disabled  
WRT1 = Off       ' Write Protection Block 1 Disabled  
WRT2 = Off       ' Write Protection Block 2 Disabled  
WRT3 = Off       ' Write Protection Block 3 Disabled  
WRTB = Off       ' Boot Block Write Protection Disabled  
WRTC = Off       ' Configuration Register Write Protection Disabled  
WRTD = Off       ' Data EEPROM Write Protection Disabled  
EBTR0 = Off      ' Table Read Protection Block 0 Disabled  
EBTR1 = Off      ' Table Read Protection Block 1 Disabled  
EBTR2 = Off      ' Table Read Protection Block 2 Disabled  
EBTR3 = Off      ' Table Read Protection Block 3 Disabled  
EBTRB = Off      ' Boot Block Table Read Protection Disabled
```

Config_End

The configs shown are for the 18F452 device and differ from device to device.

A complete list of Config fuse settings can be found in the "hlpPIC18ConfigSet.chm" file downloadable from www.microchip.com.

The fuse setting text between **Config_Start** and **Config_End** will have the preceding **Config** text added, then is passed directly to the assembler. Any errors in the fuse setting texts will result in Assembler errors being produced.

Notes

If the **Config** directive is not used within the BASIC program then default values are used. These may be found in the .ppi files within the "Includes\PPI" folder.

When using either of the **Config** directives, always use all the fuse settings for the particular PICmicro™ used. With 14-bit core (16F) devices, the compiler will always issue a reminder after the **Config** directive has been issued, however, this may be ignored if you are confident that you have assigned all the relevant fuse names.

Any fuse names that are omitted from the **Config** list will normally assume an Off or Disabled state. However, this is not always the case, and unpredictable results may occur, or the PICmicro™ may refuse to start up altogether.

Before programming the PICmicro™, always check the user configured fuse settings at programming time to ensure that the settings are correct.

Always read the datasheet for the particular PICmicro™ of interest, before using this directive.

Config1 and Config2

Some 14-bit core devices have more than one configuration area, therefore additional **Config** directives have been added. These are **Config1** and **Config2**. Their use is exactly the same as the **Config** directive, but the fuse names depend on the device used:

Example:

' Alter the fuse settings for a 16F886 device

```
Config1 HS_OSC, WDT_OFF, DEBUG_OFF, FCMEN_OFF, IESO_OFF, _  
BOR_OFF, CPD_OFF, CP_OFF, MCLR_ON, PWRTE_ON
```

```
Config2 WRT_OFF, BOR21V
```

Note that at the time of writing, all enhanced 14-bit core devices have 2 config areas.

Continue

Syntax Continue

Overview

Cause the next iteration of a **For...Next**, **While...Wend** or **Repeat...Until** loop to occur. With a **For...Next** loop, **Continue** will jump to the **Next** part. With a **While...Wend** loop, **Continue** will jump to the **While** part. With a **Repeat...Until** loop, **Continue** will jump to the **Until** part.

Example

```
' Create and display a For-Next loop's iterations, missing out number 10
Device = 18F25K20
Declare Xtal = 4

Dim Index as Byte

For Index = 0 to 19          ' Create a loop of 20 iterations
  If Index = 10 Then Continue ' Miss out number 10
  Hrsout Dec Index, 13      ' Display the counting loop
  DelayMs 100              ' Slow things down to see what's happening
Next                        ' Close the loop
```

See also : **Break**, **For...Next**, **Repeat...Until**, **While...Wend**.

Context

Syntax

Context Save {Variable, Variable}

Context Restore

Overview

Save and restore important variables and device SFRs (Special Function Registers) while inside an interrupt. **Context Restore** will also exit the interrupt and hand control back to the main program.

Operators

Variable is an optional list of user-defined variables or SFRs that will also be saved before entering the interrupt handling subroutine and restored after the interrupt has ended.

Example:

```
' Illustrate a typical use for Context Save and Context Restore
Device = 18F4520
Declare Xtal = 20

On_Hardware_Interrupt Goto ISR_Handler ' Point to the interrupt handler
Dim wTimer1 as TMR1L.Word ' Create a 16-bit Word from registers TMR1L/H

Goto Main ' Jump over any subroutines to main program
' -----
' A typical Interrupt handling subroutine
ISR_Handler:
Context Save ' Save any variables used in the interrupt
If PIR1bits_TMR1IF = 1 Then ' Is it a Timer1 overflow interrupt?
Toggle PORTB.0 ' Yes. So. Toggle PORTB.0
PIR1bits_TMR1IF = 0 ' Clear the Timer1 Overflow flag
EndIf
Context Restore ' Restore any variables and exit the interrupt
' -----
' The main program starts here
Main:
Low PORTB ' Make all of PORTB output low
'
' Setup a Timer1 interrupt
T1CONbits_RD16 = 1 ' Enable read/write of Timer1 in 16-bit mode
T1CONbits_T1CKPS1 = 0 ' \ Timer1 Prescaler to 1:4
T1CONbits_T1CKPS0 = 1 ' /
T1CONbits_T1OSCEN = 0 ' Disable External Oscillator
T1CONbits_TMR1CS = 0 ' Increment on the internal Clock
wTimer1 = 0 ' Clear Timer1
T1CONbits_TMR1ON = 1 ' Enable Timer1
PIE1bits_TMR1IE = 1 ' Enable the Timer1 overflow interrupt
INTCON1bits_PEIE = 1 ' Enable all peripheral interrupts
INTCON1bits_GIE = 1 ' Enable all interrupts

While ' Create an infinite loop
PORTB.1 = 1 ' Set PORTB.1 high
DelayMS 200 ' Wait a while
PORTB.1 = 0 ' Pull PORTB.1 low
DelayMs 200 ' Wait a while
Wend ' Do it forever
```

Notes.

When an interrupt occurs, it will immediately leave the main program and jump to the interrupt handling subroutine regardless of what the main program is doing. The main program generally has no idea that an interrupt has occurred and if it was using any of the device's resources or the compiler's system variables and the interrupt handler is doing the same, they will be altered when the main program continues, with disastrous results.

This is the reason for context saving and restoring of the compiler's internal system variables and the device's SFRs (Special Function Registers). Each compiler command generates variables for it to work upon, either for passing parameters or the actual working of the library routine. Some commands also make use of the device's SFRs, for example **FSR** or **PRODL** or **PRODH** etc...

Of course, we don't want to save every internal system variable or device SFR as this would take far too much RAM and slow down the entry and exit of the interrupt while each was saved and restored. What we want is to save and restore only the variables and SFRs that are used within the interrupt handler itself. This may be a lot or a little, or none, depending on the program within the interrupt handler subroutine.

The compiler examines the code between the **Context Save** and **Context Restore** commands and keeps a record of the internal compiler system variables and SFRs used. There are exceptions to this rule concerning SFRs which we'll deal with later.

The **Context Save** command should always be at the beginning of the interrupt handling subroutine, and this will save any variables in a specially created byte array.

Exceptions to the Rule.

Each of the compiler's commands reports internally as to which compiler system variable and SFR they use. However, this is not the case for any SFRs used as an assignment variable. For example:

```
PRODL = ByteIn1 + ByteIn2
```

It is also not the case for any **PORT** or **TRIS** registers.

For these SFRs to be saved and restored they will need to be added to the list of parameters after the **Context Save** command:

```
Context Save PRODL, PORTB, TRISB
```

See also : [On_Hardware_Interrupt](#), [On_Low_Interrupt](#).

Counter

Syntax

Variable = **Counter** *Pin*, *Period*

Overview

Count the number of pulses that appear on *pin* during *period*, and store the result in *variable*.

Operators

Variable is a user-defined variable.

Pin is a Port.Pin constant declaration i.e. PORTA.0.

Period may be a constant, variable, or expression.

Example

```
' Count the pulses that occur on PORTA.0 within a 100ms period
' and displays the results.

Dim Wrd as Word           ' Declare a word size variable
Symbol Pin = PORTA.0     ' Assign the input pin to PORTA.0
Cls
Loop:
Wrd = Counter Pin, 100   ' Variable Wrd now contains the Count
Cursor 1, 1
Print Dec Wrd, " "       ' Display the decimal result on the LCD
Goto Loop                ' Do it indefinitely
```

Notes

The resolution of *period* is in milliseconds (ms). It obtains its scaling from the oscillator declaration, **Declare Xtal**.

Counter checks the state of the pin in a concise loop, and counts the rising edge of a transition (low to high).

With a 4MHz oscillator, the pin is checked every 20us, and every 4us with a 20MHz oscillator. From this we can determine that the highest frequency of pulses that may be counted is: -

25KHz using a 4MHz oscillator.

125KHz using a 20MHz oscillator.

See also : **PulseIn**, **Rcin**.

cPtr8, cPtr16, cPtr32

Syntax

Variable = **cPtr8** (*Address*)

Variable = **cPtr16** (*Address*)

Variable = **cPtr32** (*Address*)

Overview

Indirectly read code memory using a variable to hold the 16-bit or 32-bit address. For enhanced 14-bit core devices and 18F devices only.

Operands

Variable is a user defined variable that holds the result of the indirectly addressed code memory area.

Address is a **Word** or **Dword** variable that holds the 16-bit or 32-bit address of the code memory area of interest.

Address can also post or pre increment or decrement:

- (MyAddress++) Post increment MyAddress after retrieving it's RAM location.
- (MyAddress--) Post decrement MyAddress after retrieving it's RAM location.
- (++MyAddress) Pre increment MyAddress before retrieving it's RAM location.
- (--MyAddress) Pre decrement MyAddress before retrieving it's RAM location.

cPtr8 will retrieve a value with an optional 8-bit post or pre increment or decrement.

cPtr16 will retrieve a value with an optional 16-bit post or pre increment or decrement.

cPtr32 will retrieve a value with an optional 32-bit post or pre increment or decrement.

8-bit Example.

```
'  
' Read 8-bit values indirectly from code memory  
'  
Device = 18F25K20           ' Choose an 18F device  
Declare Xtal = 16  
  
Declare Hserial_Baud = 9600 ' Set baud rate to 9600  
Declare Hserial_TXSTA = %00100000 ' Enable transmit  
'  
' Create an 8-bit code memory array  
'  
Dim CodeArray As Code = as Byte 1, 2, 3, 4, 5, 6, 7, 8, 9, 0  
Dim MyByte As Byte           ' Create a byte variable  
Dim bIndex As Byte  
Dim wAddress As Word         ' Create variable to hold 16-bit address  
Main:  
'  
' Read from code memory  
'  
wAddress = AddressOf(CodeArray) ' Load wAddress with address of memory  
While                               ' Create a loop  
    MyByte = cPtr8(wAddress++) ' Retrieve from code with post increment  
    If MyByte = 0 Then Break ' Exit when a null(0) is read from code  
    HRSOut Dec MyByte, 13 ' Transmit the byte read from code  
Wend
```

16-bit Example.

```
'  
' Read 16-bit values indirectly from code memory  
'  
Device = 18F25K20           ' Choose an 18F device  
Declare Xtal = 16  
  
Declare Hserial_Baud = 9600     ' Set baud rate to 9600  
Declare Hserial_TXSTA = %00100000 ' Enable transmit  
'  
' Create a 16-bit code memory array  
'  
Dim CodeArray As Code = as Word 100, 200, 300, 400, 500, 600, 700, 0  
Dim MyWord As Word           ' Create a word variable  
Dim bIndex As Byte  
Dim wAddress As Word         ' Create variable to hold 16-bit address
```

Main:

```
'  
' Read from code memory  
'  
wAddress = AddressOf(CodeArray) ' Load wAddress with address of memory  
While                                     ' Create a loop  
    MyWord = cPtr16(wAddress++) ' Retrieve from code with post increment  
    If MyWord = 0 Then Break      ' Exit when a null(0) is read from code  
    HRSOut Dec MyWord, 13        ' Transmit the word read from code  
Wend
```

32-bit Example.

```
'  
' Read 32-bit values indirectly from code memory  
'  
Device = 18F25K20           ' Choose an 18F device  
Declare Xtal = 16  
  
Declare Hserial_Baud = 9600     ' Set baud rate to 9600  
Declare Hserial_TXSTA = %00100000 ' Enable transmit  
'  
' Create a 32-bit code memory array  
'  
Dim CodeArray As Code = as Dword 100, 200, 300, 400, 500, 600, 700, 0  
Dim MyDword As Dword         ' Create a dword variable  
Dim bIndex As Byte  
Dim wAddress As Word         ' Create variable to hold 16-bit address
```

Main:

```
'  
' Read from code memory  
'  
wAddress = AddressOf(CodeArray) ' Load wAddress with address of memory  
While                                     ' Create a loop  
    MyDword = cPtr32(wAddress++) ' Retrieve from code with post increment  
    If MyDword = 0 Then Break      ' Exit when a null(0) is read from code  
    HRSOut Dec MyDword, 13        ' Transmit the dword read from code  
Wend
```

See also: [AddressOf](#), [Cread8](#), [Cread16](#), [Cread32](#).

Cread

Syntax

Variable = **Cread** *Address*

Overview

Read data from anywhere in code memory.

Operators

Variable is a user defined variable.

Address is a constant, variable, label, or expression that represents any valid address within code memory

Example

' Read code memory locations within the device

```
Device = 16F877
Dim Var1 as Byte
Dim Wrđ as Word
Dim Address as Word
Address = 1000           ' Address now holds the base address
Var1 = Cread 1000       ' Read 8-bit data at address 1000 into Var1
Wrđ = Cread Address + 10 ' Read data at address 1000+10
```

Notes

The **Cread** command takes advantage of the self-modifying feature that is available in the latest devices.

If a **Float** or **Dword** size variable is used as the assignment, then 32-bits will be read. If a **Word** size variable is used as the assignment, then 16-bits will be read. If a **Byte** sized variable is used as the assignment, then 8-bits will be read.

The configuration fuse setting WRTE must be enabled before **Cdata**, **Cread**, and **Cwrite** may be used, this is the default setting. This enables the self-modifying feature. If the **Config** directive is used, then the WRTE_ON fuse setting must be included in the list: -

```
Config WDT_ON, XT_OSC, WRTE_ON
```

See also : **Cdata**, **Cread8**, **Cread16**, **Cread32**, **Config**, **Cwrite**, **Dim**, **Ldata**, **Lread**, **Lread8**, **Lread16**, **Lread32**.

Cread8, Cread16, Cread32

Syntax

Variable = **Cread8** *Label* [*Offset Variable*]

or

Variable = **Cread16** *Label* [*Offset Variable*]

or

Variable = **Cread32** *Label* [*Offset Variable*]

Overview

Read an 8, 16, or 32-bit value from a **Cdata** table using an offset of *Offset Variable* and place into *Variable*, with more efficiency than using **Cread** . For device's that can access their own code memory.

Cread8 will access 8-bit values from an **Cdata** table.

Cread16 will access 16-bit values from an **Cdata** table.

Cread32 will access 32-bit values from an **Cdata** table, this also includes floating point values.

Operators

Variable is a user defined variable or an **Array**.

Label is a label name preceding the **Cdata** statement of which values will be read from.

Offset Variable can be a constant value, variable, or expression that points to the location of interest within the **Cdata** table.

Cread8 Example

```
' Extract the second value from within an 8-bit Cdata table
Device = 18F452
Dim Offset as Byte      ' Declare a Byte size variable for the offset
Dim Result as Byte     ' Declare a Byte size variable to hold the result

Cls                    ' Clear the LCD
Offset = 1             ' Point to the second value in the Ldata table
' Read the 8-bit value pointed to by Offset
Result = Cread8 Byte_Table[Offset]
Print Dec Result      ' Display the decimal result on the LCD
Stop

' Create a table containing only 8-bit values
Byte_Table: Cdata as Byte 100, 200
```

Cread16 Example

```
' Extract the second value from within a 16-bit Cdata table
Device = 18F4520
Dim Offset as Byte      ' Declare a Byte size variable for the offset
Dim Result as Word      ' Declare a Word size variable to hold the result

Cls                      ' Clear the LCD
Offset = 1               ' Point to the second value in the Ldata table
' Read the 16-bit value pointed to by Offset
Result = Cread16 WordTable[Offset]
Print Dec Result        ' Display the decimal result on the LCD
Stop

' Create a table containing only 16-bit values
WordTable: Cdata as Word 1234, 5678
```

Cread32 Example

```
' Extract the second value from within a 32-bit Cdata table
Device = 18F4520
Dim Offset as Byte      ' Declare a Byte size variable for the offset
Dim Result as Dword     ' Declare a Dword size variable to hold the result

Cls                      ' Clear the LCD
Offset = 1               ' Point to the second value in the Ldata table
' Read the 32-bit value pointed to by Offset
Result = Cread32 DwordTable[Offset]
Print Dec Result        ' Display the decimal result on the LCD
Stop

' Create a table containing only 32-bit values
DwordTable: Cdata as Dword 12340, 56780
```

Notes

Data storage in any program is of paramount importance, and although the standard **Cread** command can access multi-byte values from an **Cdata** table, it was not originally intended as such, and is more suited to accessing character data or single 8-bit values. However, the **Cread8**, **Cread16**, and **Cread32** commands are specifically written in order to efficiently read data from an **Cdata** table, and use the least amount of code space in doing so, thus increasing the speed of operation. Which means that wherever possible, **Cread** should be replaced by **Cread8**, **Cread16**, or **Cread32**.

See also :**Cdata**, **Cread**, **Dim**, **Ldata**, **Lread**, **Lread8**, **Lread16**, **Lread32**.

Cursor

Syntax

Cursor *Line, Position*

Overview

Move the cursor position on an Alphanumeric or Graphic LCD to a specified line (ypos) and position (xpos).

Operators

Line is a constant, variable, or expression that corresponds to the line (Ypos) number from 1 to maximum lines (0 to maximum lines if using a graphic LCD).

Position is a constant, variable, or expression that moves the position within the position (Xpos) chosen, from 1 to maximum position (0 to maximum position if using a graphic LCD).

Example 1

```
Dim Line as Byte
Dim Xpos as Byte
Line = 2
Xpos = 1
Cls                ' Clear the LCD
Print "HELLO"      ' Display the word "HELLO" on the LCD
Cursor Line, Xpos  ' Move the cursor to line 2, position 1
Print "WORLD"      ' Display the word "WORLD" on the LCD
```

In the above example, 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.

Example 2

```
Dim Xpos as Byte
Dim Ypos as Byte
Again:
Ypos = 1           ' Start on line 1
For Xpos = 1 to 16 ' Create a loop of 16
  Cls              ' Clear the LCD
  Cursor Ypos, Xpos ' Move the cursor to position Ypos,Xpos
  Print "*"        ' Display the character
  DelayMs 100
Next
Ypos = 2           ' Move to line 2
For Xpos = 16 to 1 Step -1 ' Create another loop, this time reverse
  Cls              ' Clear the LCD
  Cursor Ypos, Xpos ' Move the cursor to position Ypos,Xpos
  Print "*"        ' Display the character
  DelayMs 100
Next
Goto Again        ' Repeat forever
```

Example 2 displays an asterisk character moving around the perimeter of a 2-line by 16 character LCD.

See also : **Cls, Print**

Cwrite

Syntax

Cwrite Address, [Variable {, Variable...}]

Overview

Write data to anywhere in code memory on devices that support it.

Operators

Variable can be a constant, variable, or expression.

Address is a constant, variable, label, or expression that represents any valid code memory address

Example

```
' Write to memory location 2000+ within the PICmicro

Device = 16F877           ' Needs to be a 16F87x type PICmicro
Dim Var1 as Byte
Dim Wrd as Word
Dim Address as Word
Address = 2000           ' Address now holds the base address
Var1 = 234
Wrd = 1043
Cwrite Address, [10, Var1, Wrd] ' Write to address 2000 +
Org 2000
```

Notes

The **Cwrite** command takes advantage of the new self-modifying feature that is available in the newer 16F87x, and 18F series devices.

If a **Word** size variable is used as the assignment, then a 14-bit **Word** will be written. If a **Byte** sized variable is used as the assignment, then 8-bits will be written.

Because the 14-bit core devices are only capable of holding 14 bits to a **Word**, values greater than 16383 (\$3FFF) cannot be written. However, the 18F devices may hold values up to 65535 (\$FFFF).

The configuration fuse setting WRTE must be enabled before **Cdata**, **Cread**, and **Cwrite** may be used, this is the default setting. This enables the self-modifying feature. If the Config directive is used, then the WRTE_ON fuse setting must be included in the list: -

```
Config WDT_ON, XT_OSC, WRTE_ON
```

See also : **Cdata**, **Config**, **Cread**, **Cread8**, **Cread16**, **Cread32**.

Dec

Syntax

Dec *Variable*

Overview

Decrement a variable i.e. $Var1 = Var1 - 1$

Operators

Variable is a user defined variable

Example

```
Var1 = 11
Repeat
  Dec Var1
  Print Dec Var1, " "
  DelayMs 200
Until Var1 = 0
```

The above example shows the equivalent to the **For-Next** loop: -

```
For Var1 = 10 to 0 Step -1
Next
```

See also : **Inc.**

Declare

Syntax

[Declare] *code modifying directive* = *modifying value*

Overview

Adjust certain aspects of the produced code at compile time, i.e. Crystal frequency, LCD port and pins, serial baud rate etc.

Operators

code modifying directive is a set of pre-defined words. See list below.

modifying value is the value that corresponds to the action. See list below.

The **Declare** directive is an indispensable part of the compiler. It moulds the library subroutines, and passes essential user information to them.

Notes

The **Declare** directive usually alters the corresponding library subroutine at runtime. This means that once the **Declare** is added to the BASIC program, it usually cannot be Undeclared later, or changed in any way. However, there are some declares that alter the flow of code, and can be enabled and disabled throughout the BASIC listing.

Oscillator Frequency Declare.

12-bit core device XTAL values:

Declare Xtal 4, 8, 10, 12, 16, or 20.

Standard 14-bit core device XTAL values:

Declare Xtal 3, 4, 7, 8, 10, 12, 14, 16, 19, 20, 22, or 24.

Enhanced 14-bit core device XTAL values:

Declare Xtal 3, 4, 7, 8, 10, 12, 14, 16, 19, 20, 22, 24, 32 or 48.

18F device XTAL values:

Declare Xtal 3, 4, 7, 8, 10, 12, 14, 16, 19, 20, 22, 24, 25, 29, 32, 33, 40, 48, or 64.

Inform the compiler what frequency oscillator is being used.

Some commands are very dependant on the oscillator frequency, **Rsin**, **Rsout**, **DelayMs**, and **DelayUs** being just a few. In order for the compiler to adjust the correct timing for these commands, it must know what frequency crystal is being used.

The **Xtal** frequencies 3, 7, 14, 19 and 22 are for 3.58MHz, 7.2MHz, 14.32MHz, 19.66MHz, 22.1184MHz and 29.2MHz respectively.

If the **Declare** is not used in the program, then the default frequency will be of an unknown state.

Misc Declares.

Declare WatchDog = On or Off, or True or False, or 1, 0

The **WatchDog Declare** directive enables or disables the watchdog timer. It also sets the PICmicro's Config fuses for no watchdog. In addition, it removes any **CirWdt** mnemonics from the assembled code, thus producing slightly smaller programs. The default for the compiler is **WatchDog Off**, therefore, if the watchdog timer is required, then this **Declare** will need to be invoked.

The **WatchDog Declare** can be issued multiple times within the BASIC code, enabling and disabling the watchdog timer as and when required.

Declare BootLoader = On or Off, or True or False, or 1, 0

The **BootLoader Declare** directive enables or disables the special settings that a serial boot-loader requires at the start of code space. This directive is ignored if a PICmicro™ without boot-loading capabilities is targeted.

Disabling the bootloader will free a few bytes from the code produced. This doesn't seem a great deal, however, these few bytes may be the difference between a working or non-working program. The default for the compiler is **BootLoader On**

Declare Show_System_Variables = On or Off, or True or False, or 1, 0

When using the Proteus VSM to simulate BASIC code, it is sometimes beneficial to observe the behaviour of the compiler's System variables that are used for its library routines. The **Show_System_Variables Declare** enables or disables this option.

Declare Warnings = On or Off, or True or False, or 1, 0

The **Warnings Declare** directive enables or disables the compiler's warning messages. This can have disastrous results if a warning is missed or ignored, so use this directive sparingly, and at your own peril.

The **Warnings Declare** can be issued multiple times within the BASIC code, enabling and disabling the warning messages at key points in the code as and when required.

Declare Reminders = On or Off, or True or False, or 1, 0

The **Reminders Declare** directive enables or disables the compiler's reminder messages. The compiler issues a reminder for a reason, so use this directive sparingly, and at your own peril.

The **Reminders Declare** can be issued multiple times within the BASIC code, enabling and disabling the warning messages at key points in the code as and when required.

Declare Label_Bank_Resets = On or Off, or True or False, or 1, 0

The compiler has very intuitive RAM bank handling, however, if you think that an anomaly is occurring due to misplaced or mishandled RAM bank settings, you can issue this **Declare** and it will reset the RAM bank on every BASIC label, which will force the compiler to re-calculate its bank settings. If nothing else, it will reassure you that bank handling is not the cause of the problem, and you can get on with finding the cause of the programming problem. However, if it does cure a problem then please let me know and I will make sure the anomaly is fixed as quickly as possible.

Using this **Declare** will increase the size of the code produced, as it will place **Bcf** mnemonics in the case of a 12 or 14-bit core device, and a **Movlb** mnemonic in the case of an 18F device.

The **Label_Bank_Resets Declare** can be issued multiple times within the BASIC code, enabling and disabling the bank resets at key points in the code as and when required. See Line Labels for more information.

Declare Float_Display_Type = Fast or Standard

By default, the compiler uses a relatively small routine for converting floating point values to decimal, ready for **Rsout**, **Print**, **Str\$** etc. However, because of its size, it does not perform any rounding of the value first, and is only capable of converting relatively small values. i.e. approx 6 digits of accuracy. In order to produce a more accurate result, the compiler needs to use a larger routine. This is implemented by using the above **Declare**.

Using the Fast model for the above **Declare** will trigger the compiler into using the more accurate floating point to decimal routine. Note that even though the routine is larger than the standard converter, it actually operates much faster.

The compiler defaults to Standard if the **Declare** is not issued in the BASIC program.

Declare Create_Coff = On or Off, or True or False or 1, 0

When the **Create_Coff Declare** is set to **On**, the compiler produces a cof file (Common Object File). This is used for simulating the BASIC code within the MPLAB™ IDE environment or the ISIS simulator.

Declare ICD_Req = On or Off, or True or False, or 1, 0

When the **ICD_Req Declare** is set to **On**, the compiler configures itself so that the Microchip ICD2™ In-Circuit-Debugger, or PICkit2™ can be used. The ICD2™ and PICkit2™ are very invasive to the program, in so much that they require certain RAM areas for itself. This can be up to 26 bytes on some PICmicros. They also require 2 call-stack levels, so be careful when using a 14-bit core device or you may overflow the call-stack with disastrous results.

With a 14-bit core device, the top of Bank0 RAM is reserved for the ICD, for 18F devices, the RAM usage is not so noticeable because of its linear nature, but it still requires 12 bytes reserved at the end of RAM.

The list below highlights the requirements for the ICD2™ with some devices that support it.

Device	RAM Usage
P12F675	\$54 - \$5F
P12F629	\$54 - \$5F
P16F627A	\$70 - \$7F
P16F628A	\$70 - \$7F
P16F648A	\$70 - \$7F
P16F630	\$54 - \$5F
P16F676	\$54 - \$5F
P16F87	\$70 - \$7F
P16F88	\$70 - \$7F
P16F818	\$65 - \$7F
P16F819	\$65 - \$7F
P16F870	\$70 - \$7F, \$B5 - \$BF
P16F871	\$70 - \$7F, \$B5 - \$BF
P16F872	\$70 - \$7F, \$B5 - \$BF
P16F873/873A	\$74 - \$7F
P16F874/874A	\$74 - \$7F
P16F876/876A	\$70 - \$7F
P16F877/877A	\$70 - \$7F
P18F242/442	\$02F4 - \$02FF
P18F252/452	\$05F4 - \$05FF
P18F248/448	\$02F4 - \$02FF
P18F258/458	\$05F4 - \$05FF
P18F1220	\$F4 - \$FF
P18F1320	\$F4 - \$FF
P18F2220/4220	\$01F4 - \$01FF
P18F2320/4320	\$01F4 - \$01FF
P18F2331/4331	\$02F4 - \$02FF
P18F2431/4431	\$02F4 - \$02FF
P18F2680/4680	\$0CF4 - \$0CFF
P18F6520/8520	\$0EF4 - \$0EFF
P18F6620/8620	\$0EF4 - \$0EFF
P18F6720/8720	\$0EF4 - \$0EFF

Whenever ICD2™ or PICkit2™ compatibility is enabled, the compiler will automatically deduct the reserved RAM from the available RAM within the PICmicro™, therefore you must take this into account when declaring variables. Remember, there aren't as many variables available with the ICD enabled.

If the ICD is enabled along with hardware interrupts, the compiler will also reserve the RAM required for context saving and restoring. This also will be reflected in the amount of RAM available within the PICmicro™.

Note that the above list will increase as new PICmicro™ devices are released. Therefore, the help file will contain the most up to date listing of compatible devices.

Adin Declares.

Declare Adin_Res 8, 10, or 12.

Sets the number of bits in the result.

If this **Declare** is not used, then the default is the resolution of the PICmicro™ type used. For example, the new 16F87X range will result in a resolution of 10-bits, while the standard PICmicro™ types will produce an 8-bit result. Using the above **Declare** allows an 8-bit result to be obtained from the 10-bit PICmicro™ types, but not 10-bits from the 8-bit types.

Declare Adin_Tad 2_FOSC, 8_FOSC, 32_FOSC, 64_FOSC or FRC.

Sets the ADC's clock source.

All compatible PICmicros have four options for the clock source used by the ADC; 2_FOSC, 8_FOSC, and 32_FOSC, are ratios of the external oscillator, while FRC is the PICmicro's internal RC oscillator. Instead of using the predefined names for the clock source, values from 0 to 3 may be used. These reflect the settings of bits 0-1 in register ADCON0.

Care must be used when issuing this **Declare**, as the wrong type of clock source may result in poor resolution, or no conversion at all. If in doubt use FRC which will produce a slight reduction in resolution and conversion speed, but is guaranteed to work first time, every time. FRC is the default setting if the **Declare** is not issued in the BASIC listing.

Declare Adin_Stime 0 to 65535 microseconds (us).

Allows the internal capacitors to fully charge before a sample is taken. This may be a value from 0 to 65535 microseconds (us).

A value too small may result in a reduction of resolution. While too large a value will result in poor conversion speeds without any extra resolution being attained.

A typical value for **Adin_Stime** is 50 to 100. This allows adequate charge time without losing too much conversion speed.

But experimentation will produce the right value for your particular requirement. The default value if the **Declare** is not used in the BASIC listing is 50.

Busin - Busout Declares.

Declare SDA_Pin Port . Pin

Declares the port and pin used for the data line (SDA). This may be any valid port on the PICmicro™. If this declare is not issued in the BASIC program, then the default Port and Pin is PORTA.0

Declare SCL_Pin Port . Pin

Declares the port and pin used for the clock line (SCL). This may be any valid port on the PICmicro™. If this declare is not issued in the BASIC program, then the default Port and Pin is PORTA.1

Declare Slow_Bus On - Off or 1 - 0

Slows the bus speed when using an oscillator higher than 4MHz.

The standard speed for the I²C bus is 100KHz. Some devices use a higher bus speed of 400KHz. If you use an 8MHz or higher oscillator, the bus speed may exceed the devices specs, which will result in intermittent writes or reads, or in some cases, none at all. Therefore, use this **Declare** if you are not sure of the device's spec. The datasheet for the device used will inform you of its bus speed.

Declare Bus_SCL On - Off, 1 - 0 or True - False

Eliminates the necessity for a pull-up resistor on the SCL line.

The I²C protocol dictates that a pull-up resistor is required on both the SCL and SDA lines, however, this is not always possible due to circuit restrictions etc, so once the **Bus_SCL On Declare** is issued at the top of the program, the resistor on the SCL line can be omitted from the circuit. The default for the compiler if the **Bus_SCL Declare** is not issued, is that a pull-up resistor is required.

Hbusin - Hbusout Declare.

Declare Hbus_Bitrate Constant 100, 400, 1000 etc.

The standard speed for the I²C bus is 100KHz. Some devices use a higher bus speed of 400KHz. The above **Declare** allows the I²C bus speed to be increased or decreased. Use this **Declare** with caution, as too high a bit rate may exceed the device's specs, which will result in intermittent transactions, or in some cases, no transactions at all. The datasheet for the device used will inform you of its bus speed. The default bit rate is the standard 100KHz.

Hserin, Hserout, Hrsin and Hrsout Declares.

Declare Hserial_Baud Constant value

Sets the BAUD rate that will be used to receive a value serially. The baud rate is calculated using the **Xtal** frequency declared in the program. The default baud rate if the **Declare** is not included in the program listing is 2400 baud.

Declare Hserial_RCSTA Constant value (0 to 255)

Hserial_RCSTA, sets the respective PICmicro™ hardware register RCSTA, to the value in the **Declare**. See the Microchip data sheet for the device used for more information regarding this register.

Declare Hserial_TXSTA Constant value (0 to 255)

Hserial_TXSTA, sets the respective PICmicro™ hardware register, TXSTA, to the value in the **Declare**. See the Microchip data sheet for the device used for more information regarding this register. The TXSTA register BRGH bit (bit 2) controls the high speed mode for the baud rate generator. Certain baud rates at certain oscillator speeds require this bit to be set to operate properly. To do this, set **Hserial_TXSTA** to a value of \$24 instead of the default \$20. Refer to the Microchip data sheet for the hardware serial port baud rate tables and additional information.

Declare **Hserial_Parity** Odd or Even

Enables/Disables parity on the serial port. For **Hrsin**, **Hrsout**, **Hserin** and **Hserout**. The default serial data format is 8N1, 8 data bits, no parity bit and 1 stop bit. 7E1 (7 data bits, even parity, 1 stop bit) or 7O1 (7 data bits, odd parity, 1 stop bit) may be enabled using the **Hserial_Parity** declare.

```
Declare Hserial_Parity = Even      ' Use if even parity desired
Declare Hserial_Parity = Odd      ' Use if odd parity desired
```

Declare **Hserial_Clear** On or Off

Clear the overflow error bit before commencing a read.

Because the hardware serial port only has a 2-byte input buffer, it can easily overflow if characters are not read from it often enough. When this occurs, the USART stops accepting any new characters, and requires resetting. This overflow error can be reset by strobing the CREN bit within the RCSTA register. Example: -

```
RCSTA.4 = 0
RCSTA.4 = 1
```

OR

```
Clear RCSTA.4
Set RCSTA.4
```

Alternatively, the **Hserial_Clear** declare can be used to automatically clear this error, even if no error occurred. However, the program will not know if an error occurred while reading, therefore some characters may be lost.

```
Declare Hserial_Clear = On
```

USART2 Declares for use with **Hrsin2**, **Hserin2**, **Hrsout2** and **Hserout2**.

Declare **Hserial2_Baud** Constant value

Sets the BAUD rate that will be used to transmit a value serially. The baud rate is calculated using the **Xtal** frequency declared in the program. The default baud rate if the **Declare** is not included in the program listing is 2400 baud.

Declare **Hserial2_RCSTA** Constant value (0 to 255)

Hserial2_RCSTA, sets the respective hardware register RCSTA2, to the value in the **Declare**. See the Microchip data sheet for the device used for more information regarding this register. Refer to the upgrade manual pages for a description of the RCSTA2 register.

Declare **Hserial2_TXSTA** Constant value (0 to 255)

Hserial2_TXSTA, sets the respective hardware register, TXSTA2, to the value in the **Declare**. See the Microchip data sheet for the device used for more information regarding this register. The TXSTA register BRGH2 bit (bit 2) controls the high speed mode for the baud rate generator. Certain baud rates at certain oscillator speeds require this bit to be set to operate properly. To do this, set **Hserial2_TXSTA** to a value of \$24 instead of the default \$20. Refer to the Microchip data sheet for the hardware serial port baud rate tables and additional information. Refer to the upgrade manual pages for a description of the TXSTA2 register.

Declare **Hserial2_Parity** Odd or Even

Enables/Disables parity on the serial port. For **Hrsout2**, **Hrsin2**, **Hserout2** and **Hserin2**. The default serial data format is 8N1, 8 data bits, no parity bit and 1 stop bit. 7E1 (7 data bits, even parity, 1 stop bit) or 7O1 (7 data bits, odd parity, 1 stop bit) may be enabled using the **Hserial2_Parity** declare.

```
Declare Hserial2_Parity = Even   ' Use if even parity desired
Declare Hserial2_Parity = Odd    ' Use if odd parity desired
```

Declare **Hserial2_Clear** On or Off

Clear the overflow error bit before commencing a read.

Because the hardware serial port only has a 2-byte input buffer, it can easily overflow if characters are not read from it often enough. When this occurs, the USART stops accepting any new characters, and requires resetting. This overflow error can be reset by strobing the CREN bit within the RCSTA2 register. Example: -

```
RCSTA2.4 = 0
RCSTA2.4 = 1
```

or

```
Clear RCSTA2.4
Set   RCSTA2.4
```

Alternatively, the **Hserial2_Clear** declare can be used to automatically clear this error, even if no error occurred. However, the program will not know if an error occurred while reading, therefore some characters may be lost.

```
Declare Hserial2_Clear = On
```

Hpwm Declares.

Some devices, such as the PIC16F62x, and PIC18F4xx, have alternate pins that may be used for **Hpwm**. The following **Declares** allow the use of different pins: -

```
Declare CCP1_Pin Port.Pin ' Select Hpwm port and bit for CCP1 module (ch 1)
Declare CCP2_Pin Port.Pin ' Select Hpwm port and bit for CCP2 module (ch 2)
```

Alphanumeric (Hitachi) LCD Print Declares.

Declare LCD_DTPin Port . Pin

Assigns the Port and Pins that the LCD's DT lines will attach to.

The LCD may be connected to the PICmicro™ using either a 4-bit bus or an 8-bit bus. If an 8-bit bus is used, all 8 bits must be on one port. If a 4-bit bus is used, it must be connected to either the bottom 4 or top 4 bits of one port. For example: -

```
Declare LCD_DTPin PORTB.4    ' Used for 4-line interface.  
Declare LCD_DTPin PORTB.0    ' Used for 8-line interface.
```

In the above examples, PORTB is only a personal preference. The LCD's DT lines can be attached to any valid port on the PICmicro™. If the **Declare** is not used in the program, then the default Port and Pin is PORTB.4, which assumes a 4-line interface.

Declare LCD_ENPin Port . Pin

Assigns the Port and Pin that the LCD's EN line will attach to. This also assigns the graphic LCD's EN pin, however, the default value remains the same as for the alphanumeric type, so this will require changing.

If the **Declare** is not used in the program, then the default Port and Pin is PORTB.2.

Declare LCD_RSPin Port . Pin

Assigns the Port and Pins that the LCD's RS line will attach to. This also assigns the graphic LCD's RS pin, however, the default value remains the same as for the alphanumeric type, so this will require changing.

If the **Declare** is not used in the program, then the default Port and Pin is PORTB.3.

Declare LCD_Interface 4 or 8

Inform the compiler as to whether a 4-line or 8-line interface is required by the LCD.

If the **Declare** is not used in the program, then the default interface is a 4-line type.

Declare LCD_Lines 1, 2, or 4

Inform the compiler as to how many lines the LCD has.

LCD's come in a range of sizes, the most popular being the 2 line by 16 character types. However, there are 4-line types as well. Simply place the number of lines that the particular LCD has into the declare.

If the **Declare** is not used in the program, then the default number of lines is 2.

Declare LCD_CommandUS 1 to 65535

Time to wait (in microseconds) between commands sent to the LCD.

If the **Declare** is not used in the program, then the default delay is 2000us (2ms).

Declare LCD_DataUs 1 to 255

Time to wait (in microseconds) between data sent to the LCD.

If the **Declare** is not used in the program, then the default delay is 50us.

Graphic LCD Declares.

Declare LCD_Type 0 or 1 or 2, **Alpha** or **Graphic** or **Samsung** or **Toshiba**

Inform the compiler as to the type of LCD that the **Print** command will output to. If **Graphic**, **Samsung** or 1 is chosen then any output by the **Print** command will be directed to a graphic LCD based on the Samsung KS0108 chipset. A value of 2, or the text **Toshiba**, will direct the output to a graphic LCD based on the Toshiba T6963 chipset. A value of 0 or **Alpha**, or if the **Declare** is not issued, will target the standard Hitachi alphanumeric LCD type

Targeting the graphic LCD will also enable commands such as **Plot**, **UnPlot**, **LCDread**, **LCDwrite**, **Pixel**, **Box**, **Circle** and **Line**.

Samsung KS0108 Graphic LCD specific Declares.

Declare LCD_DTPort Port

Assign the port that will output the 8-bit data to the graphic LCD.

If the **Declare** is not used, then the default port is PORTB.

Declare LCD_RWPin Port . Pin

Assigns the Port and Pin that the graphic LCD's RW line will attach to.

If the **Declare** is not used in the program, then the default Port and Pin is PORTC.0.

Declare LCD_CS1Pin Port . Pin

Assigns the Port and Pin that the graphic LCD's CS1 line will attach to.

If the **Declare** is not used in the program, then the default Port and Pin is PORTC.0.

Declare LCD_CS2Pin Port . Pin

Assigns the Port and Pin that the graphic LCD's CS2 line will attach to.

If the **Declare** is not used in the program, then the default Port and Pin is PORTC.0.

Declare Internal_Font On - Off, 1 or 0

The graphic LCD's that are compatible with Proton are non-intelligent types, therefore, a separate character set is required. This may be in one of two places, either externally, in an I²C eeprom, or internally in a **Cdata** table.

If an external font is chosen, the I²C eeprom must be connected to the specified SDA and SCL pins (as dictated by **Declare SDA_Pin** and **Declare SCL_Pin**).

If an internal font is chosen, it must be on a PICmicro™ device that has self modifying code features, such as the 16F87X, or 18F range.

The **Cdata** table that contains the font must have a label, named `Font_Table`: preceding it. For example: -

```
Font_Table:  Cdata $7E, $11, $11, $11, $7E, $0      ' Chr "A"  
            Cdata $7F, $49, $49, $49, $36, $0      ' Chr "B"  
            { rest of font table }
```

The font table may be anywhere in memory, however, it is best placed after the main program code.

If the **Declare** is omitted from the program, then an external font is the default setting.

Declare Font_Addr 0 to 7

Set the slave address for the I²C eeprom that contains the font.

When an external source for the font is chosen, it may be on any one of 8 eeproms attached to the I²C bus. So as not to interfere with any other eeproms attached, the slave address of the eeprom carrying the font code may be chosen.

If the **Declare** is omitted from the program, then address 0 is the default slave address of the font eeprom.

Declare GLCD_CS_Invert On - Off, 1 or 0

Some graphic LCD types have inverters on their CS lines. Which means that the LCD displays left hand data on the right side, and vice-versa. The **GLCD_CS_Invert Declare**, adjusts the library LCD handling library subroutines to take this into account.

Declare GLCD_Strobe_Delay 0 to 65535 us (microseconds).

Create a delay of n microseconds between strobing the EN line of the graphic LCD. This can help noisy, or badly decoupled circuits overcome random bits appearing on the LCD. The default if the **Declare** is not used in the BASIC program is a delay of 0.

Toshiba T6963 Graphic LCD specific Declares.

Declare LCD_DTPort Port

Assign the port that will output the 8-bit data to the graphic LCD.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_WRPin Port . Pin

Assigns the Port and Pin that the graphic LCD's WR line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_RDPin Port . Pin

Assigns the Port and Pin that the graphic LCD's RD line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_CEPin Port . Pin

Assigns the Port and Pin that the graphic LCD's CE line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_CDPin Port . Pin

Assigns the Port and Pin that the graphic LCD's CD line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_RSTPin Port . Pin

Assigns the Port and Pin that the graphic LCD's RST line will attach to.

The LCD's RST (Reset) **Declare** is optional and if omitted from the BASIC code the compiler will not manipulate it. However, if not used as part of the interface, you must set the LCD's RST pin high for normal operation.

Declare LCD_X_Res 0 to 255

LCD displays using the T6963 chipset come in varied screen sizes (resolutions). The compiler must know how many horizontal pixels the display consists of before it can build its library sub-routines.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_Y_Res 0 to 255

LCD displays using the T6963 chipset come in varied screen sizes (resolutions). The compiler must know how many vertical pixels the display consists of before it can build its library sub-routines.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_Font_Width 6 or 8

The Toshiba T6963 graphic LCDs have two internal font sizes, 6 pixels wide by eight high, or 8 pixels wide by 8 high. The particular font size is chosen by the LCD's FS pin. Leaving the FS pin floating or bringing it high will choose the 6 pixel font, while pulling the FS pin low will choose the 8 pixel font. The compiler must know what size font is required so that it can calculate screen and RAM boundaries.

Note that the compiler does not control the FS pin and it is down to the circuit layout whether or not it is pulled high or low. There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_RAM_Size 1024 to 65535

Toshiba graphic LCDs contain internal RAM used for Text, Graphic or Character Generation. The amount of RAM is usually dictated by the display's resolution. The larger the display, the more RAM is normally present. Standard displays with a resolution of 128x64 typically contain 4096 bytes of RAM, while larger types such as 240x64 or 190x128 typically contain 8192 bytes or RAM. The display's datasheet will inform you of the amount of RAM present.

If this **Declare** is not issued within the BASIC program, the default setting is 8192 bytes.

Declare LCD_Text_Pages 1 to n

As mentioned above, Toshiba graphic LCDs contain RAM that is set aside for text, graphics or characters generation. In normal use, only one page of text is all that is required, however, the compiler can re-arrange its library subroutines to allow several pages of text that is continuous. The amount of pages obtainable is directly proportional to the RAM available within the LCD itself. Larger displays require more RAM per page, therefore always limit the amount of pages to only the amount actually required or unexpected results may be observed as text, graphic and character generator RAM areas merge.

This **Declare** is purely optional and is usually not required. The default is 3 text pages if this **Declare** is not issued within the BASIC program.

Declare LCD_Graphic_Pages 1 to n

Just as with text, the Toshiba graphic LCDs contain RAM that is set aside for graphics. In normal use, only one page of graphics is all that is required, however, the compiler can re-arrange its library subroutines to allow several pages of graphics that is continuous. The amount of pages obtainable is directly proportional to the RAM available within the LCD itself. Larger displays require more RAM per page, therefore always limit the amount of pages to only the amount actually required or unexpected results may be observed as text, graphic and character generator RAM areas merge.

This **Declare** is purely optional and is usually not required. The default is 1 graphics page if this **Declare** is not issued within the BASIC program.

Declare LCD_Text_Home_Address 0 to n

The RAM within a Toshiba graphic LCD is split into three distinct uses, text, graphics and character generation. Each area of RAM must not overlap or corruption will appear on the display as one uses the other's assigned space. The compiler's library subroutines calculate each area of RAM based upon where the text RAM starts. Normally the text RAM starts at address 0, however, there may be occasions when it needs to be set a little higher in RAM. The order of RAM is; Text, Graphic, then Character Generation.

This **Declare** is purely optional and is usually not required. The default is the text RAM starting at address 0 if this **Declare** is not issued within the BASIC program.

Keypad Declare.

Declare Keypad_Port Port

Assigns the Port that the keypad is attached to.

The keypad routine requires pull-up resistors, therefore, the best Port for this device is PORTB which comes equipped with internal pull-ups. If the **Declare** is not used in the program, then PORTB is the default Port.

Rsin - Rsout Declares.

Declare Rsout_Pin Port . Pin

Assigns the Port and Pin that will be used to output serial data from the **Rsout** command. This may be any valid port on the PICmicro™.

If the **Declare** is not used in the program, then the default Port and Pin is PORTB.0.

Declare Rsin_Pin Port . Pin

Assigns the Port and Pin that will be used to input serial data by the **Rsin** command. This may be any valid port on the PICmicro™.

If the **Declare** is not used in the program, then the default Port and Pin is PORTB.1.

Declare Rsout_Mode True or Inverted or 1, 0

Sets the serial mode for the data transmitted by **Rsout**. This may be inverted or true. Alternatively, a value of 1 may be substituted to represent inverted, and 0 for true.

If the **Declare** is not used in the program, then the default mode is inverted.

Declare Rsin_Mode True or Inverted or 1, 0

Sets the serial mode for the data received by **Rsin**. This may be inverted or true. Alternatively, a value of 1 may be substituted to represent inverted, and 0 for true.

If the **Declare** is not used in the program, then the default mode is inverted.

Declare Serial_Baud 0 to 65535 bps (baud)

Informs the **Rsin** and **Rsout** routines as to what baud rate to receive and transmit data.

Virtually any baud rate may be transmitted and received (within reason), but there are standard bauds, namely: -

300, 600, 1200, 2400, 4800, 9600, and 19200.

When using a 4MHz crystal, the highest baud rate that is reliably achievable is 9600. However, an increase in the oscillator speed allows higher baud rates to be achieved, including 38400 baud.

If the **Declare** is not used in the program, then the default baud is 9600.

Declare Rsout_Pace 0 to 65535 microseconds (us)

Implements a delay between characters transmitted by the **Rsout** command.

On occasion, the characters transmitted serially are in a stream that is too fast for the receiver to catch, this results in missed characters. To alleviate this, a delay may be implemented between each individual character transmitted by **Rsout**.

If the **Declare** is not used in the program, then the default is no delay between characters.

Declare Rsin_Timeout 0 to 65535 milliseconds (ms)

Sets the time, in ms, that **Rsin** will wait for a start bit to occur.

Rsin waits in a tight loop for the presence of a start bit. If no timeout parameter is issued, then it will wait forever.

The Rsin command has the option of jumping out of the loop if no start bit is detected within the time allocated by timeout.

If the **Declare** is not used in the program, then the default timeout value is 10000ms which is 10 seconds.

Serin - Serout Declare.

If communications are with existing software or hardware, its speed and mode will determine the choice of baud rate and mode. In general, 7-bit/even-parity (7E) mode is used for text, and 8-bit/no-parity (8N) for byte-oriented data. Note: the most common mode is 8-bit/no-parity, even when the data transmitted is just text. Most devices that use a 7-bit data mode do so in order to take advantage of the parity feature. Parity can detect some communication errors, but to use it you lose one data bit. This means that incoming data bytes transferred in 7E (even-parity) mode can only represent values from 0 to 127, rather than the 0 to 255 of 8N (no-parity) mode.

The compiler's serial commands **Serin** and **Serout** have the option of still using a parity bit with 4 to 8 data bits. This is through the use of a **Declare**: -

With parity disabled (the default setting): -

```
Declare Serial_Data 4 ' Set Serin and Serout data bits to 4
Declare Serial_Data 5 ' Set Serin and Serout data bits to 5
Declare Serial_Data 6 ' Set Serin and Serout data bits to 6
Declare Serial_Data 7 ' Set Serin and Serout data bits to 7
Declare Serial_Data 8 ' Set Serin and Serout data bits to 8 (default)
```

With parity enabled: -

```
Declare Serial_Data 5 ' Set Serin and Serout data bits to 4
Declare Serial_Data 6 ' Set Serin and Serout data bits to 5
Declare Serial_Data 7 ' Set Serin and Serout data bits to 6
Declare Serial_Data 8 ' Set Serin and Serout data bits to 7 (default)
Declare Serial_Data 9 ' Set Serin and Serout data bits to 8
```

Serial_Data data bits may range from 4 bits to 8 (the default if no **Declare** is issued). Enabling parity uses one of the number of bits specified.

Declaring **Serial_Data** as 9 allows 8 bits to be read and written along with a 9th parity bit.

Parity is a simple error-checking feature. When a serial sender is set for even parity (the mode the compiler supports) it counts the number of 1s in an outgoing byte and uses the parity bit to make that number even. For example, if it is sending the 7-bit value: %0011010, it sets the parity bit to 1 in order to make an even number of 1s (four).

The receiver also counts the data bits to calculate what the parity bit should be. If it matches the parity bit received, the serial receiver assumes that the data was received correctly. Of course, this is not necessarily true, since two incorrectly received bits could make parity seem correct when the data was wrong, or the parity bit itself could be bad when the rest of the data was correct.

Many systems that work exclusively with text use 7-bit/ even-parity mode. For example, to receive one data byte through bit-0 of PORTA at 9600 baud, 7E, inverted:

Shin - Shout Declare.

Declare Shift_DelayUs 0 - 65535 microseconds (us)

Extend the active state of the shift clock.

The clock used by **Shin** and **Shout** runs at approximately 45KHz dependent on the oscillator. The active state is held for a minimum of 2 microseconds. By placing this declare in the program, the active state of the clock is extended by an additional number of microseconds up to 65535 (65.535 milliseconds) to slow down the clock rate.

If the **Declare** is not used in the program, then the default is no clock delay.

Compact Flash Interface Declares

There are several declares that need to be manipulated when interfacing to a Compact Flash card. There are the obvious port pins, but there are also some declares that optimise or speed up access to the card.

Declare **CF_DTPort** Port

This **Declare** assigns the Compact Flash card's data lines. The data line consists of 8-bits so it is only suitable for ports that contain 8-bits such as PORTB, PORTC, PORTD etc.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare **LCD_ADPort** Port

This **Declare** assigns the Compact Flash card's address lines. The address line consists of 3-bits, but A0 of the compact flash card must be attached to bit-0 of whatever port is used. For example, if the Compact Flash card's address lines were attached to PORTA of the PICmicro™, then A0 of the CF card must attach to PORTA.0, A1 of the CF card must attach to PORTA.1, and A2 of the CF card must attach to PORTA.2.

The CF access commands will mask the data before transferring it to the particular port that is being used so that the rest of it's pins are not effected. PORTE is perfect for the address lines as it contains only 3 pins on a 40-pin device, and the compiler can make full use of this by using the **CF_ADPort_Mask** declare.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare CF_ADPort_Mask = On or Off, or True or False, or 1, 0

Both the **CF_Write** and **CF_Sector** commands write to the Compact Flash card's address lines. However, these only contain 3-bits, so the commands need to ensure that the other bits of the PICmicro's Port are not effected. This is accomplished by masking the unwanted data before transferring it to the address lines. This takes a little extra code space, and thus a little extra time to accomplish. However, there are occasions when the condition of the other bits on the Port are not important, or when a Port is used that only has 3-bits to it. i.e. PORTE with a 40-pin device. Issuing the **CF_ADPort_Mask Declare** and setting it False, will remove the masking mnemonics, thus reducing code used and time taken.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare CF_RDYPin Port . Pin

Assigns the Compact Flash card's RDY/BSY line.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare CF_OEPin Port . Pin

Assigns the Compact Flash card's OE line.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare CF_WEPin Port . Pin

Assigns the Compact Flash card's WE line.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare CF_CD1Pin Port . Pin

Assigns the Compact Flash card's CD1 line. The CD1 line is not actually used by any of the commands, but is set to input if the declare is issued in the BASIC program. The CD1 line is used to indicate whether the card is inserted into its socket.

There is no default setting for this **Declare**.

Declare CF_RSTPin Port . Pin

Assigns the Compact Flash card's RESet line. The RESet line is not essential for interfacing to a Compact Flash card, but is useful if a clean power up is required. If the declare is not issued in the BASIC program, all reference to it is removed from the **CF_Init** command. If the RESet line is not used for the card, ensure that it is tied to ground.

There is no default setting for this **Declare**.

Declare CF_CE1Pin Port . Pin

Assigns the Compact Flash card's CE1 line. As with the RESet line, the CE1 line is not essential for interfacing to a Compact Flash card, but is useful when multiplexing pins, as the card will ignore all commands when the CE1 line is set high. If the declare is not issued in the BASIC program, all reference to it is removed from the **CF_Init** command. If the CE1 line is not used for the card, ensure that it is tied to ground.

There is no default setting for this **Declare**.

Declare CF_Read_Write_Inline = On or Off, or True or False, or 1, 0

Sometimes, speed is of the essence when accessing a Compact Flash card, especially when interfacing to the new breed of card which is 40 times faster than the normal type. Because of this, the compiler has the ability to create the code used for the **CF_Write** and **CF_Read** commands inline, which means it does not call its library subroutines, and can tailor itself when reading or writing **Word**, **Dword**, or **Float** variables. However, this comes at a price of code memory, as each command is stretched out for speed, not optimisation. It also means that the inline type of commands are really only suitable for the higher speed Compact Flash cards.

If the **Declare** is not used in the BASIC program, the default is not to use **INLINE** commands.

DelayCs

Syntax

DelayCs Length

Overview

Delay execution for an amount of instruction cycles.

Operators

- **Length** can only be a constant with a value from 1 to 1000.

Example

```
DelayCs 100           ' Delay for 100 cycles
```

Notes

DelayCs is oscillator independent, as long as you inform the compiler of the crystal frequency to use, using the **Declare** directive.

The length of a given instruction cycle is determined by the oscillator frequency. For example, running the microcontroller at it's default speed of 64MHz will result in an instruction cycle of 62.5ns (nano seconds).

Because of code memory paging overheads, **DelayCs** is only available when using enhanced 14-bit core or 18F devices.

See also : **DelayUs**, **DelayMs**, **Sleep**, **Snooze**.

DelayMs

Syntax

DelayMs *Length*

Overview

Delay execution for *length* x milliseconds (ms). Delays may be up to 65535ms (65.535 seconds) long.

Operators

Length can be a constant, variable, or expression.

Example

```
Declare Xtal = 4
Dim Var1 as Byte
Dim Wrd1 as Word
Var1 = 50
Wrd1 = 1000
DelayMs 100           ' Delay for 100ms
DelayMs Var1          ' Delay for 50ms
DelayMs Wrd1          ' Delay for 1000ms
DelayMs Wrd1 + 10     ' Delay for 1010ms
```

Notes

DelayMs is oscillator independent, as long as you inform the compiler of the crystal frequency to use, using the Declare directive.

See also : [DelayUs](#), [Sleep](#), [Snooze](#).

DelayUs

Syntax

DelayUs *Length*

Overview

Delay execution for *length* x microseconds (us). Delays may be up to 65535us (65.535 milliseconds) long.

Operators

Length can be a constant, variable, or expression.

Example

```
Declare Xtal = 20
Dim Var1 as Byte
Dim Wrld as Word
Var1 = 50
Wrld = 1000
DelayUs 1           ' Delay for 1us
DelayUs 100         ' Delay for 100us
DelayUs Var1        ' Delay for 50us
DelayUs Wrld        ' Delay for 1000us
DelayUs Wrld + 10   ' Delay for 1010us
```

Notes

DelayUs is oscillator independent, as long as you inform the compiler of the crystal frequency to use, using the **Xtal** directive.

If a constant is used as *length*, then delays down to 1us can be achieved, however, if a variable is used as *length*, then there's a minimum delay time depending on the frequency of the crystal used: -

Crystal Freq	Minimum Delay
4MHz	24us
8MHz	12us
10MHz	8us
16MHz	5us
20MHz	2us
24MHz	2us
25MHz	2us
32MHz	2us
33MHz	2us
40MHz	2us
48MHz	2us
64MHz	2us

See also : **Declare, DelayMs, Sleep, Snooze**

Device

Syntax

Device *Device number*

Overview

Inform the compiler which microcontroller is being used.

Operators

Device number can be a 12-bit, 14-bit, enhanced 14-bit, or 18F device.

Example

```
Device = 16F877      ' Produce code for a 16F877 device
```

or

```
Device = 16F684      ' Produce code for a 16F684 device
```

or

```
Device = 12F508      ' Produce code for a 12-bit core 12F508 device
```

or

```
Device = 18F4520     ' Produce code for a 18F4520 device
```

Device should be the first command placed in the program.

If the **Device** directive is not used in the BASIC program, the code produced will default to the ever-popular (but now outdated) 16F84 device.

For an up-to-date list of compatible devices refer to the compiler's PPI folder.

Dig

Syntax

Variable = **Dig** *Value*, *Digit number*

Overview

Returns the value of a decimal digit.

Operators

Value is an unsigned constant, 8-bit, 16-bit, 32-bit variable or expression, from which the *digit number* is to be extracted.

Digit number is a constant, variable, or expression, that represents the digit to extract from *value*. (0 - 4 with 0 being the rightmost digit).

Example

```
Dim Var1 as Byte
Dim Var2 as Byte
Var1 = 124
Var2 = Dig Var1, 1 ' Extract the second digit's value
Print Dec Var2    ' Display the value, which is 2
```

Dim

Syntax

Dim *Variable* **as** *Size*

or

Dim *Label* **as** **Code** = *comma delimited data*

Overview

Declare a variable or alias or code memory table.

Operators

Variable can be any alphanumeric character or string.

Size is the physical size of the variable, it may be **Bit**, **Byte**, **Word**, **Dword**, **SByte**, **SWord**, **SDword**, **Float**, or **String**.

Label is a valid label name that will be associated with a code memory table.

Example

```
' Declare different sized variables
Dim Var1 as Byte      ' Declare an unsigned 8-bit Byte variable
Dim Wrld1 as Word    ' Declare an unsigned 16-bit Word variable
Dim Dwrld1 as Dword  ' Declare an unsigned 32-bit Dword variable

Dim sVar1 as SByte   ' Declare a signed 8-bit SByte variable
Dim sWrld1 as SWord  ' Declare a signed 16-bit SWord variable
Dim sDwrld1 as SDword ' Declare a signed 32-bit SDword variable

Dim BitVar as Bit    ' Declare a 1-bit Bit variable
Dim Flt as Float     ' Create a 32-bit floating point variable
Dim StrnG as String*20 ' Create a 20 character string variable
Dim MyCode as Code = 1,2,3,4,5,6,7 ' Place 7 bytes in code memory
```

Notes

Any variable that is declared without the '**as**' text after it, will assume an 8-bit **Byte** type.

Dim should be placed near the beginning of the program. Any references to variables not declared or before they are declared may, in some cases, produce errors.

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

```
Dim MyVar as Byte
or
Dim MY_Var as Word
or
Dim My_Var2 as Bit
```

Variable names may start with an underscore, but must not start with a number. They can be no more than 32 characters long. Any characters after this limit will cause a syntax error.

Dim 2MyVar is **not** allowed.

Variable names are not case sensitive, which means that the variable: -

```
Dim MYVar  
Is the same as...  
Dim MYVar
```

Dim can also be used to create Alias's to other variables: -

```
Dim Var1 as Byte      ' Declare a Byte sized variable  
Dim Var_Bit as Var1.1 ' Var_Bit now represents Bit-1 of Var1
```

Alias's, as in the case of constants, do not require any RAM space, because they point to a variable, or part of a variable that has already been declared.

RAM space required.

Each type of variable requires differing amounts of RAM memory for its allocation. The list below illustrates this.

- **String** Requires the specified length of characters + 1.
- **Float** Requires 4 bytes of RAM.
- **Dword** Requires 4 bytes of RAM.
- **SDword** Requires 4 bytes of RAM.
- **Word** Requires 2 bytes of RAM.
- **SWord** Requires 2 bytes of RAM.
- **Byte** Requires 1 byte of RAM.
- **SByte** Requires 1 byte of RAM.
- **Bit** Requires 1 byte of RAM for every 8 Bit variables used.

Each type of variable may hold a different minimum and maximum value.

- **String** type variables are only useable with 18F and enhanced 14-bit core devices, and can hold a maximum of 255 characters.
- **Bit** type variables may hold a 0 or a 1. These are created 8 at a time, therefore declaring a single **Bit** type variable in a program will not save RAM space, but it will save code space, as **Bit** type variables produce the most efficient use of code for comparisons etc.
- **Byte** type variables may hold an unsigned value from 0 to 255, and are the usual work horses of most programs. Code produced for **Byte** sized variables is very low compared to signed or unsigned **Word**, **DWord** or **Float** types, and should be chosen if the program requires faster, or more efficient operation.
- **SByte** type variables may hold a 2¹⁵ complemented signed value from -128 to +127. Code produced for **SByte** sized variables is very low compared to **SWord**, **Float**, or **SDword** types, and should be chosen if the program requires faster, or more efficient operation. However, code produced is usually larger for signed variables than unsigned types.
- **Word** type variables may hold an unsigned value from 0 to 65535, which is usually large enough for most applications. It still uses more memory than an 8-bit byte variable, but not nearly as much as a **Dword** or **SDword** type.

- **SWord** type variables may hold a 2^{18} complemented signed value from -32768 to +32767, which is usually large enough for most applications. **SWord** type variables will use more code space for expressions and comparisons, therefore, only use signed variables when required.
- **Dword** type variables may hold an unsigned value from 0 to 4294967295 making this the largest of the variable family types. This comes at a price however, as **Dword** calculations and comparisons will use more code space within the microcontroller Use this type of variable sparingly, and only when necessary.
- **SDword** type variables may hold a 2^{18} complemented signed value from -2147483648 to +2147483647, also making this the largest of the variable family types. This comes at a price however, as **SDword** expressions and comparisons will use more code space than a regular **Dword** type. Use this type of variable sparingly, and only when necessary.
- **Float** type variables may theoretically hold a value from -1e37 to +1e38, but because of the 32-bit architecture of the compiler, a maximum and minimum value should be thought of as -2147483646.999 to +2147483646.999 making this the most versatile of the variable family types. However, more so than **Dword** types, this comes at a price as floating point expressions and comparisons will use more code space within the microcontroller. Use this type of variable sparingly, and only when strictly necessary. Smaller floating point values usually offer more accuracy.

There are modifiers that may also be used with variables. These are **HighByte**, **LowByte**, **Byte0**, **Byte1**, **Byte2**, **Byte3**, **Word0**, **Word1**, **SHighByte**, **SLowByte**, **SByte0**, **SByte1**, **SByte2**, **SByte3**, **SWord0**, and **SWord1**,

Word0, **Word1**, **Byte2**, **Byte3**, **SWord0**, **SWord1**, **SByte2**, and **SByte3** may only be used in conjunction with 32-bit **Dword** or **SDword** type variables.

HighByte and **Byte1** are one and the same thing, when used with a **Word** or **SWord** type variable, they refer to the unsigned High byte of a **Word** or **SWord** type variable: -

```
Dim Wrd as Word          ' Declare an unsigned Word variable
Dim Wrd_Hi as Wrd.HighByte
' Wrd_Hi now represents the unsigned high byte of variable Wrd
```

Variable Wrd_Hi is now accessed as a **Byte** sized type, but any reference to it actually alters the high byte of Wrd.

SHighByte and **SByte1** are one and the same thing, when used with a **Word** or **SWord** type variable, they refer to the signed High byte of a **Word** or **SWord** type variable: -

```
Dim Wrd as SWord        ' Declare a signed Word variable
Dim Wrd_Hi as Wrd.SHighByte
' Wrd_Hi now represents the signed high byte of variable Wrd
```

Variable Wrd_Hi is now accessed as an **SByte** sized type, but any reference to it actually alters the high byte of Wrd.

However, if **Byte1** is used in conjunction with a **Dword** type variable, it will extract the second byte. **HighByte** will still extract the high byte of the variable, as will **Byte3**. If **SByte1** is used in conjunction with an **SDword** type variable, it will extract the signed second byte. **SHighByte** will still extract the signed high byte of the variable, as will **SByte3**.

The same is true of **LowByte**, **Byte0**, **SLowByte** and **SByte0**, but they refer to the unsigned or signed Low Byte of a **Word** or **SWord** type variable: -

```
Dim Wrđ as Word      ' Declare an unsigned Word variable
Dim Wrđ_Lo as Wrđ.LowByte
' Wrđ_Lo now represents the low byte of variable Wrđ
```

Variable **Wrđ_Lo** is now accessed as a **Byte** sized type, but any reference to it actually alters the low byte of **Wrđ**.

The modifier **Byte2** will extract the 3rd unsigned byte from a 32-bit **Dword** or **SDword** type variable as an alias. Likewise **Byte3** will extract the unsigned high byte of a 32-bit variable.

```
Dim Dwd as Dword     ' Declare a 32-bit unsigned variable named Dwd
Dim Part1 as Dwd.Byte0 ' Alias unsigned Part1 to the low byte of Dwd
Dim Part2 as Dwd.Byte1 ' Alias unsigned Part2 to the 2nd byte of Dwd
Dim Part3 as Dwd.Byte2 ' Alias unsigned Part3 to the 3rd byte of Dwd
Dim Part4 as Dwd.Byte3 ' Alias unsigned Part3 to the high (4th) byte of
Dwd
```

The modifier **SByte2** will extract the 3rd signed byte from a 32-bit **Dword** or **SDword** type variable as an alias. Likewise **SByte3** will extract the signed high byte of a 32-bit variable.

```
Dim sDwd as SDword   ' Declare a 32-bit signed variable named sDwd
Dim sPart1 as sDwd.SByte0 ' Alias signed Part1 to the low byte of sDwd
Dim sPart2 as sDwd.SByte1 ' Alias signed Part2 to the 2nd byte of sDwd
Dim sPart3 as sDwd.SByte2 ' Alias signed Part3 to the 3rd byte of sDwd
Dim sPart4 as sDwd.SByte3 ' Alias signed Part3 to the 4th byte of sDwd
```

The **Word0** and **Word1** modifiers extract the unsigned low word and high word of a **Dword** or **SDword** type variable, and is used the same as the **Byten** modifiers.

```
Dim Dwd as Dword     ' Declare a 32-bit unsigned variable named Dwd
Dim Part1 as Dwd.Word0 ' Alias unsigned Part1 to the low word of Dwd
Dim Part2 as Dwd.Word1 ' Alias unsigned Part2 to the high word of Dwd
```

The **SWord0** and **SWord1** modifiers extract the signed low word and high word of a **Dword** or **SDword** type variable, and is used the same as the **SByten** modifiers.

```
Dim sDwd as SDword   ' Declare a 32-bit signed variable named sDwd
Dim sPart1 as sDwd.SWord0 ' Alias Part1 to the low word of sDwd
Dim sPart2 as sDwd.SWord1 ' Alias Part2 to the high word of sDwd
```

RAM space for variables is allocated within the microcontroller in the order that they are placed in the BASIC code. For example: -

```
Dim Var1 as Byte
Dim Var2 as Byte
```

Places Var1 first, then Var2: -

```
Var1 equ n
Var2 equ n
```

This means that on a device with more than one RAM Bank, the first n variables will always be in Bank0 (the value of n depends on the specific PICmicro™ used).

The position of the variable within Banks is usually of little importance if BASIC code is used, however, if assembler routines are being implemented, always assign any variables used within them first.

Problems may also arise if a **Word**, **Dword** or **Float** variable crosses a RAM bank boundary. If this happens, a warning message will be displayed in the error window. Most of the time, this will not cause any problems, however, to err on the side of caution, try and ensure that **Word**, **Dword** or **Float** type variables are fully inside a Bank. This is easily accomplished by placing a dummy **Byte** variable before the offending **Word**, **Dword** or **Float** type variable, or relocating the offending variable within the list of **Dim** statements.

See Also : [Aliases](#), [Declaring Arrays](#), [Floating Point Math](#), [Symbol](#), [Creating and using Strings](#) .

Disable

Disable software interrupt processing that was previously Enabled following this instruction.

Disable and **Enable**, and **Resume** are not actually commands in the truest sense of the word, but flags that the compiler uses internally. They do not produce any code.

```
Device = 16F877
OPTION_REG = %00000111
INTCON = %00100000
Symbol LED = PORTD.0
' Enable software interrupts, and point to interrupt handler
On Interrupt Goto My_Int
```

Fin:

```
DelayMs 1
Goto Fin
```

```
Disable          ' Disable interrupts in the handler
```

My_Int:

```
Toggle LED      ' Toggle an LED when interrupted
Resume          ' Return to main program
Enable         ' Enable interrupts after the handler
```

Note.

Software interrupts are a remnant from earlier compiler versions and are not recommended for new applications. See **Managed Hardware Interrupts** for a better method of interrupt handling.

See also : **Software Interrupts in BASIC, Enable, Resume.**

DTMFout

Syntax

DTMFout *Pin*, { *OnTime* }, { *OffTime*, } [*Tone* {, *Tone*...}]

Overview

Produce a DTMF Touch Tone sequence on *Pin*.

Operators

Pin is a Port.Bit constant that specifies the I/O pin to use. This pin will be set to output during generation of tones and set to input after the command is finished.

OnTime is an optional variable, constant, or expression (0 - 65535) specifying the duration, in ms, of the tone. If the *OnTime* parameter is not used, then the default time is 200ms

OffTime is an optional variable, constant, or expression (0 - 65535) specifying the length of silent delay, in ms, after a tone (or between tones, if multiple tones are specified). If the *OffTime* parameter is not used, then the default time is 50ms

Tone may be a variable, constant, or expression (0 - 15) specifying the DTMF tone to generate. Tones 0 through 11 correspond to the standard layout of the telephone keypad, while 12 through 15 are the fourth-column tones used by phone test equipment and in some radio applications.

Example

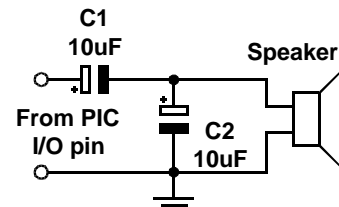
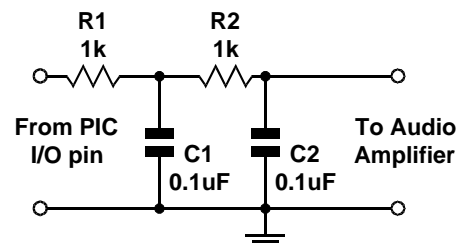
```
DTMFout PORTA.0, [ 7, 4, 9, 9, 9, 0 ] ' Call Crownhill.
```

If the PICmicro™ was connected to the phone line correctly, the above command would dial 666-709. If you wanted to slow down the dialling in order to break through a noisy phone line or radio link, you could use the optional *OnTime* and *OffTime* values: -

'Set the OnTime to 500ms and OffTime to 100ms

```
DTMFout PORTA.0, 500, 100, [7, 4, 9, 9, 9, 0] ' Call Crownhill Slowly.
```

Notes DTMF tones are used to dial a telephone, or remotely control pieces of radio equipment. The PICmicro™ can generate these tones digitally using the **DTMFout** command. However, to achieve the best quality tones, a higher crystal frequency is required. A 4MHz type will work but the quality of the sound produced will suffer. The circuits illustrate how to connect a speaker or audio amplifier to hear the tones produced.



The PICmicro™ is a digital device, however, DTMF tones are analogue waveforms, consisting of a mixture of two sine waves at different audio frequencies. So how can a digital device generate an analogue output? The PICmicro™ creates and mixes two sine waves mathematically, then uses the resulting stream of numbers to control the duty cycle of an extremely fast pulse-width modulation (Pwm) routine. Therefore, what's actually being produced from the I/O pin is a rapid stream of pulses. The purpose of the filtering arrangements illustrated above is to smooth out the high-frequency Pwm, leaving behind only the lower frequency audio. You should keep this in mind if you wish to interface the PICmicro's DTMF output to radios and other equipment that could be adversely affected by the presence of high-frequency noise on the input. Make sure to filter the DTMF output scrupulously. The circuits above are only a foundation; you may want to use an active low-pass filter with a cut-off frequency of approximately 2KHz.

Edata

Syntax

Edata *Constant1* { ,...*Constantn* etc }

Overview

Places constants or strings directly into the on-board eeprom memory of compatible PICmicro's

Operators

Constant1, **Constantn** are values that will be stored in the on-board eeprom. When using an **Edata** statement, all the values specified will be placed in the eeprom starting at location 0. The **Edata** statement does not allow you to specify an eeprom address other than the beginning location at 0. To specify a location to write or read data from the eeprom other than 0 refer to the **Eread**, **Ewrite** commands.

Example

' Stores the values 1000,20,255,15, and the ASCII values for ' H','e','l','l','o' in the eeprom starting at memory position 0.

```
Edata 1000, 20, $FF, %00001111, "Hello"
```

Notes

16-bit, 32-bit and floating point values may also be placed into eeprom memory. These are placed LSB first (Lowest Significant Byte). For example, if 1000 is placed into an **Edata** statement, then the order is: -

```
Edata 1000
```

In eeprom it looks like 232, 03

Alias's to constants may also be used in an **Edata** statement: -

```
symbol Alias = 200
```

```
Edata Alias, 120, 254, "Hello World"
```

Addressing an Edata table.

Eeprom data starts at address 0 and works up towards the maximum amount that the PICmicro™ will allow. However, it is rarely the case that the information stored in eeprom memory is one continuous piece of data. Eeprom memory is normally used for storage of several values or strings of text, so a method of accessing each piece of data is essential. Consider the following piece of code: -

```
Edata "HELLO"
```

```
Edata "WORLD"
```

Now we know that eeprom memory starts at 0, so the text "HELLO" must be located at address 0, and we also know that the text "HELLO" is built from 5 characters with each character occupying a byte of eeprom memory, so the text "WORLD" must start at address 5 and also contains 5 characters, so the next available piece of eeprom memory is located at address 10. To access the two separate text strings we would need to keep a record of the start and end address's of each character placed in the tables.

Counting the amount of eeprom memory used by each piece of data is acceptable if only a few **Edata** tables are used in the program, but it can become tedious if multiple values and strings are needing to be stored, and can lead to program glitches if the count is wrong.

Placing an identifying name before the **Edata** table will allow the compiler to do the byte counting for you. The compiler will store the eeprom address associated with the table in the identifying name as a constant value. For example: -

```
Hello_Text Edata "HELLO"  
World_Text Edata "WORLD"
```

The name Hello_Text is now recognised as a constant with the value of 0, referring to address 0 that the text string "HELLO" starts at. The World_Text is a constant holding the value 5, which refers to the address that the text string "WORLD" starts at.

Note that the identifying text *must* be located on the same line as the **Edata** directive or a syntax error will be produced. It must also not contain a postfix colon as does a line label or it will be treated as a line label. Think of it as an alias name to a constant.

Any **Edata** directives *must* be placed at the head of the BASIC program as is done with Symbols, so that the name is recognised by the rest of the program as it is parsed. There is no need to jump over **Edata** directives as you have to with **Ldata** or **Cdata**, because they do not occupy code memory, but reside in high *Data* memory.

The example program below illustrates the use of eeprom addressing.

```
' Display two text strings held in eeprom memory  
  
Include "Proton_4.Inc" ' Demo on a Proton development board  
Dim Char as Byte ' Holds the character read from eeprom  
Dim Charpos as Byte ' Holds the address within eeprom memory  
  
' Create a string of text in eeprom memory. null terminated  
HELLO Edata "HELLO ",0  
' Create another string of text in eeprom memory. null terminated  
WORLD Edata "WORLD",0  
  
DelayMs 100 ' Wait for the LCD to stabilise  
Cls ' Clear the LCD  
Charpos = HELLO ' Point Charpos to the start of text "HELLO"  
Gosub DisplayText ' Display the text "HELLO"  
Charpos = WORLD ' Point Charpos to the start of text "WORLD"  
Gosub DisplayText ' Display the text "WORLD"  
Stop ' We're all done  
  
' Subroutine to read and display the text held at the address in Charpos  
DisplayText:  
While ' Create an infinite loop  
Char = Eread Charpos ' Read the eeprom data  
If Char = 0 Then Break ' Exit when null found  
Print Char ' Display the character  
Inc Charpos ' Move up to the next address  
Wend ' Close the loop  
Return ' Exit the subroutine
```

Formatting an Edata table.

Sometimes it is necessary to create a data table with a known format for its values. For example all values will occupy 4 bytes of data space even though the value itself would only occupy 1 or 2 bytes.

```
Edata 100000, 10000, 1000, 100, 10, 1
```

The above line of code would produce an uneven data space usage, as each value requires a different amount of data space to hold the values. 100000 would require 4 bytes of eeprom space, 10000 and 1000 would require 2 bytes, but 100, 10, and 1 would only require 1 byte.

Reading these values using **Eread** would cause problems because there is no way of knowing the amount of bytes to read in order to increment to the next valid value.

The answer is to use formatters to ensure that a value occupies a predetermined amount of bytes.

These are: -

Byte
Word
Dword
Float

Placing one of these formatters before the value in question will force a given length.

```
Edata Dword 100000, Dword 10000, Dword 1000, Dword 100, Dword 10, Dword 1
```

Byte will force the value to occupy one byte of eeprom space, regardless of its value. Any values above 255 will be truncated to the least significant byte.

Word will force the value to occupy 2 bytes of eeprom space, regardless of its value. Any values above 65535 will be truncated to the two least significant bytes. Any value below 255 will be padded to bring the memory count to 2 bytes.

Dword will force the value to occupy 4 bytes of eeprom space, regardless of its value. Any value below 65535 will be padded to bring the memory count to 4 bytes. The line of code shown above uses the **Dword** formatter to ensure all the values in the **Edata** table occupy 4 bytes of eeprom space.

Float will force a value to its floating point equivalent, which always takes up 4 bytes of eeprom space.

If all the values in an **Edata** table are required to occupy the same amount of bytes, then a single formatter will ensure that this happens.

```
Edata as Dword 100000, 10000, 1000, 100, 10, 1
```

The above line has the same effect as the formatter previous example using separate **Dword** formatters, in that all values will occupy 4 bytes, regardless of their value. All four formatters can be used with the **as** keyword.

The example below illustrates the formatters in use.

```
' Convert a Dword value into a string array
' Using only BASIC commands
' Similar principle to the Str$ command

Include "Proton_4.Inc"
Dim P10 as Dword      ' Power of 10 variable
Dim Cnt as Byte
Dim J as Byte

Dim Value as Dword    ' Value to convert
Dim String1[11] as Byte ' Holds the converted value
Dim Ptr as Byte       ' Pointer within the Byte array

DelayMs 100           ' Wait for the LCD to stabilise
Cls                   ' Clear the LCD
Clear                 ' Clear all RAM before we start
Value = 1234576       ' Value to convert
Gosub DwordToStr     ' Convert Value to string
Print Str String1    ' Display the result
Stop

'-----
' Convert a Dword value into a string array
' Value to convert is placed in 'Value'
' Byte array 'String1' is built up with the ASCII equivalent

DwordToStr:
Ptr = 0
J = 0
Repeat
  P10 = Eread J * 4
  Cnt = 0

  While Value >= P10
    Value = Value - P10
    Inc Cnt
  Wend
  If Cnt <> 0 Then
    String1[Ptr] = Cnt + "0"
    Inc Ptr
  EndIf
  Inc J
Until J > 8
String1[Ptr] = Value + "0"
Inc Ptr
String1[Ptr] = 0      ' Add the null to terminate the string
Return

' Edata table is formatted for all 32 bit values.
' Which means each value will require 4 bytes of eeprom space
Edata as Dword 1000000000, 100000000, 10000000, 1000000, 100000, _
               10000, 1000, 100, 10
```

Label names as pointers in an Edata table.

If a label's name is used in the list of values in an **Edata** table, the labels address will be used. This is useful for accessing other tables of data using their address from a lookup table. See example below.

```
' Display text from two Cdata tables
' Based on their address located in a separate table
Include "Proton_4.Inc"      ' Use a 14-bit core device

Dim Address as Word
Dim DataByte as Byte

DelayMs 100                ' Wait for the LCD to stabilise
Cls                        ' Clear the LCD
Address = Eread 0          ' Locate the address of the first string
While                      ' Create an infinite loop
    DataByte = Cread Address ' Read each character from the Cdata string
    If DataByte = 0 Then ExitLoop ' Exit if null found
    Print DataByte         ' Display the character
    Inc Address            ' Next character
Wend                       ' Close the loop
ExitLoop:
Cursor 2,1                ' Point to line 2 of the LCD
Address = Eread 2          ' Locate the address of the second string
While                      ' Create an infinite loop
    DataByte = Cread Address ' Read each character from the Cdata string
    If DataByte = 0 Then ExitLoop2 ' Exit if null found
    Print DataByte         ' Display the character
    Inc Address            ' Next character
Wend                       ' Close the loop
ExitLoop2:
Stop

' Table of address's located in eeprom memory
Edata as Word String1, String2
String1:
Cdata "HELLO",0
String2:
Cdata "WORLD",0
```

See also : Eread, Ewrite.

Enable

Enable software interrupt processing that was previously Disabled following this instruction.

Disable and **Enable**, and **Resume** are not actually commands in the truest sense of the word, but flags that the compiler uses internally. They do not produce any code.

```
Device = 16F877
OPTION_REG = %00000111
INTCON = %00100000
Symbol LED = PORTD.0
' Enable software interrupts, and point to interrupt handler
  On Interrupt Goto My_Int

Fin:
  DelayMs 1
  Goto Fin

  Disable          ' Disable interrupts in the handler
My_Int:
  Toggle LED      ' Toggle an LED when interrupted
  Resume          ' Return to main program
  Enable          ' Enable interrupts after the handler
```

Note.

Software interrupts are a remnant from earlier compiler versions and are not recommended for new applications. See **Managed Hardware Interrupts** for a better method of interrupt handling.

See also : **Software Interrupts in BASIC, Disable, Resume.**

Software Interrupts in BASIC

Although the most efficient method of using an interrupt is in assembler, hardware interrupts and BASIC are poor bedfellows. By far the easiest way to write an interrupt handler is to write it in BASIC, in combination with the **On Interrupt** statement. This is not the same as the compiler's **On_Interrupt** statement, which initiates a Hardware interrupt. **On Interrupt** (two separate words.. **On Interrupt**) informs the compiler to activate its internal interrupt handling and to jump to the BASIC interrupt handler as soon as it's capable, after receiving an interrupt. However, there's no such thing as a free lunch, and there are some penalties to pay for the ease of use that this method brings.

The statement **On_Hardware_Interrupt** are also recognised by the compiler in order to clarify which type of interrupt is being implemented.

When **On Interrupt** is used, the compiler simply flags that the interrupt has happened and immediately goes back to what it was doing, before it was rudely interrupted. Unlike a hardware interrupt, it does not immediately jump to the interrupt handler. And since the compiler's commands are non re-entrant, there could be a considerable delay before the interrupt is actually handled.

For example, if the program has just started to execute a **DelayMs 2000** command when an interrupt occurs, the compiler will flag the interrupt and continue with the delay. It could be as much as 2 seconds later before the interrupt handler is executed. Any time critical routines dependant on the interrupt occurring regularly will be ruined. For example, multiplexing seven segment display.

To minimise the above problem, use only statements that don't take long to execute. For example, instead of **DelayMs 2000**, use **DelayMs 1** in a **For..Next**, or **Repeat..Until** loop. This will allow the compiler to complete each command more quickly and handle any awaiting interrupts: -

```
For Var1 = 0 to 199 : DelayMs 1 : Next ' Delay for 200ms
```

If interrupt processing needs to occur more regularly, then there is no choice but to use a hardware interrupt, with all it's quirks.

Exactly what happens when **On Interrupt** is used is this: A short interrupt handler is placed at location 4 in the PICmicro™. This interrupt handler is simply a **Return**. What this does is send the program back to what it was doing before the interrupt occurred. It does not require any processor context saving. What it doesn't do is re-enable Global Interrupts as happens when using a **Retfie** instruction.

A Call to a short subroutine is placed before each command in the BASIC program once an **On Interrupt** statement is encountered. This short subroutine checks the state of the Global Interrupt Enable bit (GIE). If it's off, an interrupt is awaiting so it vectors to the users interrupt handler. Which is essentially a BASIC subroutine.

If it is still set, the program continues with the next BASIC statement, after which, the GIE bit is checked again, and so forth.

Note.

Software interrupts are a remnant from earlier compiler versions and are not recommended for new applications. See **Managed Hardware Interrupts** for a better method of interrupt handling.

See also : **Enable, Disable, Resume.**

End

Syntax **End**

Overview

The **End** statement stops compilation of source, and creates an infinite loop.

Notes

End stops the PICmicro™ processing by placing it into a continuous loop. The port pins remain the same and the device is placed in low power mode.

See also : **Stop, Sleep, Snooze.**

Eread

Syntax

Variable = **Eread** *Address*

Overview

Read information from the on-board eeprom available on some PICmicro™ types.

Operators

Variable is a user defined variable.

Address is a constant, variable, or expression, that contains the address of interest within eeprom memory.

Example

```
Device = 16F877           ' A device with on-board eeprom
Dim Var1 as Byte
Dim Wrld as Word
Dim Dwrld as Dword

Edata 10, 354, 123456789 ' Place some data into the eeprom
Var1 = Eread 0           ' Read the 8-bit value from address 0
Wrld = Eread 1           ' Read the 16-bit value from address 1
Dwrld = Eread 3          ' Read the 32-bit value from address 3
```

Notes

If a **Float**, or **Dword** type variable is used as the assignment variable, then 4-bytes will be read from the eeprom. Similarly, if a **Word** type variable is used as the assignment variable, then a 16-bit value (2-bytes) will be read from eeprom, and if a **Byte** type variable is used, then 8-bits will be read. To read an 8-bit value while using a **Word** sized variable, use the **LowByte** modifier: -

```
Wrld.LowByte = Eread 0 ' Read an 8-bit value
Wrld.HighByte = 0      ' Clear the high byte of Wrld
```

If a 16-bit (**Word**) size value is read from the eeprom, the address must be incremented by two for the next read. Also, if a **Float** or **Dword** type variable is read, then the address must be incremented by 4.

Most of the Flash PICmicro™ types have a portion of memory set aside for storage of information. The amount of memory is specific to the individual PICmicro™ type, some, such as the 16F84, has 64 bytes, the 16F877 device has 256 bytes, and some of the 18F devices have upwards of 512 bytes.

Eeprom memory is non-volatile, and is an excellent place for storage of long-term information, or tables of values.

Reading data with the **Eread** command is almost instantaneous, but writing data to the eeprom can take up to 10ms per byte.

See also : **Edata, Ewrite**

Ewrite

Syntax

Ewrite Address, [Variable {, Variable...etc }]

Overview

Write information to the on-board eeprom available on some PICmicro™ types.

Operators

Address is a constant, variable, or expression, that contains the address of interest within eeprom memory.

Variable is a user defined variable.

Example

```
Device = 16F628           ' A device with on-board eeprom
Dim Var1 as Byte
Dim Wrd1 as Word
Dim Address as Byte
Var1 = 200
Wrd1= 2456
Address = 0               ' Point to address 0 within the eeprom
Ewrite Address, [Wrd, Var1] ' Write a 16-bit then an 8-bit value
```

Notes

If a **Dword** type variable is used, then a 32-bit value (4-bytes) will be written to the eeprom. Similarly, if a **Word** type variable is used, then a 16-bit value (2-bytes) will be written to eeprom, and if a **Byte** type variable is used, then 8-bits will be written. To write an 8-bit value while using a **Word** sized variable, use the **LowByte** modifier: -

```
Ewrite Address, [ Wrd.LowByte, Var1 ]
```

If a 16-bit (**Word**) size value is written to the eeprom, the address must be incremented by two before the next write: -

```
For Address = 0 to 64 Step 2
  Ewrite Address, [Wrd]
Next
```

Most of the Flash PICmicro™ types have a portion of memory set aside for storage of information. The amount of memory is specific to the individual PICmicro™ type, some, such as the 16F84, has 64 bytes, while the newer 16F877, and 18FXXX devices have 256 bytes.

Eeprom memory is non-volatile, and is an excellent place for storage of long-term information, or tables of values.

Writing data with the **Ewrite** command can take up to 10ms per byte, but reading data from the eeprom is almost instantaneous,.

See also : **Edata, Eread**

For...Next...Step

Syntax

```
For Variable = Startcount to Endcount [ Step { Stepval } ]  
{code body}  
Next
```

Overview

The **For...Next** loop is used to execute a statement, or series of statements a predetermined amount of 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.

Startcount 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. This does not have to be an actual number, it could be the contents of another variable, or an expression.

Stepval is an optional constant or variable by which the *variable* increases or decreases with each trip through the For-Next loop. If *startcount* is larger than *endcount*, then a minus sign must precede *stepval*.

Example 1

```
' Display in decimal, all the values of Wrd within an upward loop  
Dim Wrd as Word  
  
For Wrd = 0 to 2000 Step 2      ' Perform an upward loop  
    Print Dec Wrd , " "      ' Display the value of Wrd  
Next                          ' Close the loop
```

Example 2

```
' Display in decimal, all the values of Wrd within a downward loop  
Dim Wrd as Word  
  
For Wrd = 2000 to 0 Step -2    ' Perform a downward loop  
    Print Dec Wrd , " "      ' Display the value of Wrd  
Next                          ' Close the loop
```

Example 3

```
' Display in decimal, all the values of DwrD within a downward loop  
Dim DwrD as Dword  
  
For DwrD = 200000 to 0 Step -200 ' Perform a downward loop  
    Print Dec DwrD , " "      ' Display the value of DwrD  
Next                          ' Close the loop
```


Example 4

' Display all of Wrd1 using a expressions as parts of the For-Next construct

```
Dim Wrd1 as Word
Dim Wrd2 as Word

Wrd2 = 1000

For Wrd1= Wrd2 + 10 to Wrd2 + 1000 ' Perform a loop
    Print Dec Wrd1, " "           ' Display the value of Wrd1
Next                               ' Close the loop
```

Notes

You may have noticed from the above examples, that no variable is present after the **Next** command. A variable after **Next** is purely optional.

For-Next loops may be nested as deeply as the memory on the PICmicro™ will allow. To break out of a loop you may use the **Goto** command without any ill effects, which is exactly what the **Break** comamnd does: -

```
For Var1 = 0 to 20                ' Create a loop of 21
    If Var1 = 10 Then Goto BreakOut ' Break out of loop when Var1 is 10
Next                               ' Close the loop
```

```
BreakOut:
    Stop
```

See also : **While...Wend, Repeat...Until.**

Freqout

Syntax

Freqout *Pin, Period, Freq1 {, Freq2}*

Overview

Generate one or two sine-wave tones, of differing or the same frequencies, for a specified period.

Operators

Pin is a Port-Bit combination that specifies which I/O pin to use.

Period may be a variable, constant, or expression (0 - 65535) specifying the amount of time to generate the tone(s).

Freq1 may be a variable, constant, or expression (0 - 32767) specifying frequency of the first tone.

Freq2 may be a variable, constant, or expression (0 - 32767) specifying frequency of the second tone. When specified, two frequencies will be mixed together on the same I/O pin.

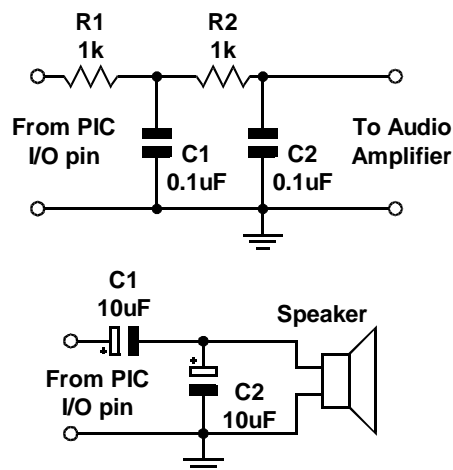
Example

```
' Generate a 2500Hz (2.5KHz) tone for 1 second (1000 ms) on bit 0 of PORTA.
  Freqout PORTA.0, 1000, 2500

' Play two tones at once for 1000ms. One at 2.5KHz, the other at 3KHz.
  Freqout PORTA.0, 1000, 2500, 30000
```

Notes

Freqout generates one or two sine waves using a pulse-width modulation algorithm. **Freqout** will work with a 4MHz crystal, however, it is best used with higher frequency crystals, and operates accurately with a 20MHz crystal. The raw output from **Freqout** requires filtering, to eliminate most of the switching noise. The circuits shown below will filter the signal in order to play the tones through a speaker or audio amplifier.



The two circuits shown above, work by filtering out the high-frequency Pwm used to generate the sine waves. **Freqout** works over a very wide range of frequencies (0 to 32767KHz) so at the upper end of its range, the Pwm filters will also filter out most of the desired frequency. You may need to reduce the values of the parallel capacitors shown in the circuit, or to create an active filter for your application.

Example 2

' Play a tune using Freqout to generate the notes

```
Device = 16F877
Declare Xtal = 20

Dim Loop as Byte      ' Counter for notes.
Dim Freq1 as Word    ' Frequency1.
Dim Freq2 as Word    ' Frequency2

Symbol C = 2092      ' C note
Symbol D = 2348      ' D note
Symbol E = 2636      ' E note
Symbol G = 3136      ' G note
Symbol R = 0         ' Silent pause.
Symbol Pin = PORTA.0 ' Sound output pin

ADCON1 = 7          ' Set PORTA and PORTE to all digital
Loop = 0
Repeat              ' Create a loop for 29 notes within the LookUpL table.
  Freq1 = LookUpL Loop, [E,D,C,D,E,E,E,R,D,D,D,_
                        R,E,G,G,R,E,D,C,D,E,E,E,E,D,D,E,D,C]
  If Freq1 = 0 Then
    Freq2 = 0
  Else
    Freq2 = Freq1 - 8
  EndIf
  Freqout Pin, 225, Freq1, Freq2
  Inc Loop
Until Loop > 28
```

See also : **DTMFout, Sound, Sound2.**

GetBit

Syntax

Variable = **GetBit** *Variable*, *Index*

Overview

Examine a bit of a variable, or register.

Operators

Variable is a user defined variable.

Index is a constant, variable, or expression that points to the bit within *Variable* that requires examining.

Example

```
' Examine and display each bit of variable ExVar
```

```
Device = 16F877
```

```
Declare Xtal = 4
```

```
Dim ExVar as Byte
```

```
Dim Index as Byte
```

```
Dim Var1 as Byte
```

```
ExVar = %10110111
```

```
Again:
```

```
Cls
```

```
Print At 1,1,Bin8 ExVar
```

```
Cursor 2,1
```

```
For Index = 7 to 0 Step -1
```

```
    Var1 = GetBit ExVar,Index
```

```
    Print Decl Var1
```

```
    DelayMs 100
```

```
Next
```

```
Goto Again
```

```
' Display the original variable
```

```
' Position the cursor at line 2
```

```
' Create a loop for 8 bits
```

```
' Examine each bit of ExVar
```

```
' Display the binary result
```

```
' Slow things down to see what's happening
```

```
' Close the loop
```

```
' Do it forever
```

See also : **ClearBit, LoadBit, SetBit.**

Gosub

Syntax

Gosub *Label*

or

Gosub *Label* [*Variable*, {*Variable*, *Variable*... etc}], *Receipt Variable*

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 instruction following the **Gosub** that called it and continues execution from that point.

If using an 18F device, parameters can be pushed onto a software stack before the call is made, and a variable can be popped from the stack before continuing execution of the next commands.

Operators

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

Variable is a user defined variable of type **Bit**, **Byte**, **Word**, **Dword**, **Float**, **String**, **Array** or **Constant** value, that will be pushed onto the stack before the call to a subroutine is performed.

Receipt Variable is a user defined variable of type **Bit**, **Byte**, **Word**, **Dword**, **Float**, **String** or **Array** that will hold a value popped from the stack after the subroutine has returned.

Example 1

```
' Implement a standard subroutine call
Goto Start      ' Jump over the subroutines
SubA: { subroutine A code
.....
.....
}
Return

SubB: { subroutine B code
.....
.....
}
Return

' Actual start of the main program
Start:
Gosub SubA
Gosub SubB
Stop
```

Example 2

```
' Call a subroutine with parameters
Device = 18F452      ' Stack only suitable for 18F devices
Declare Stack_Size = 20 ' Create a small stack capable of holding 20 bytes

Dim Wrd1 as Word      ' Create a Word variable
Dim Wrd2 as Word      ' Create another Word variable
Dim Receipt as Word   ' Create a variable to hold result

Wrd1 = 1234           ' Load the Word variable with a value
Wrd2 = 567            ' Load the other Word variable with a value
' Call the subroutine and return a value
Gosub AddThem [Wrd1, Wrd2], Receipt
Print Dec Receipt    ' Display the result as decimal
Stop

' Subroutine starts here. Add two parameters passed and return the result
AddThem:
Dim AddWrd1 as Word  ' Create two uniquely named variables
Dim AddWrd2 as Word

Pop AddWrd2           ' Pop the last variable pushed
Pop AddWrd1           ' Pop the first variable pushed
AddWrd1 = AddWrd1 + AddWrd2 ' Add the values together
Return AddWrd1        ' Return the result of the addition
```

In reality, what's happening with the **Gosub** in the above program is simple, if we break it into its constituent events: -

```
Push Wrd1
Push Wrd2
Gosub AddThem
Pop Receipt
```

Notes

Only one parameter can be returned from the subroutine, any others will be ignored.

If a parameter is to be returned from a subroutine but no parameters passed to the subroutine, simply issue a pair of empty square braces: -

```
Gosub Label [ ], Receipt
```

The same rules apply for the parameters as they do for **Push**, which is after all, what is happening.

Proton allows any amount of **Gosubs** in a program, but the 14-bit PICmicro™ architecture only has an 8-level return address stack, which only allows 8 **Gosubs** to be nested. The compiler only ever uses a maximum of 4-levels for its library subroutines, therefore do not use more than 4 **Gosubs** within subroutines. The 18F devices however, have a 28-level return address stack which allows any combination of up to 28 **GosubS** to occur.

A subroutine must always end with a **Return** command.

What is a Stack?

All microprocessors and most microcontrollers have access to a Stack, which is an area of RAM allocated for temporary data storage. But this is sadly lacking on a PICmicro™ device. However, the 18F devices have an architecture and low-level mnemonics that allow a Stack to be created and used very efficiently.

A stack is first created in high memory by issuing the **Stack_Size Declare**.

```
Declare Stack_Size = 40
```

The above line of code will reserve 40 bytes at the top of RAM that cannot be touched by any BASIC command, other than **Push** and **Pop**. This means that it is a safe place for temporary variable storage.

Taking the above line of code as an example, we can examine what happens when a variable is pushed on to the 40 byte stack, and then popped off again.

First the RAM is allocated. For this explanation we will assume that a 18F452 PICmicro™ device is being used. The 18F452 has 1536 bytes of RAM that stretches linearly from address 0 to 1535. Reserving a stack of 40 bytes will reduce the top of memory so that the compiler will only see 1495 bytes (1535 - 40). This will ensure that it will not inadvertently try and use it for normal variable storage.

Pushing.

When a **Word** variable is pushed onto the stack, the memory map would look like the diagram below: -

Top of MemoryEmpty RAM.....	Address 1535
	~	~
	~	~
Empty RAM.....	Address 1502
Empty RAM.....	Address 1501
	Low Byte address of Word variable	Address 1496
Start of Stack	High Byte address of Word variable	Address 1495

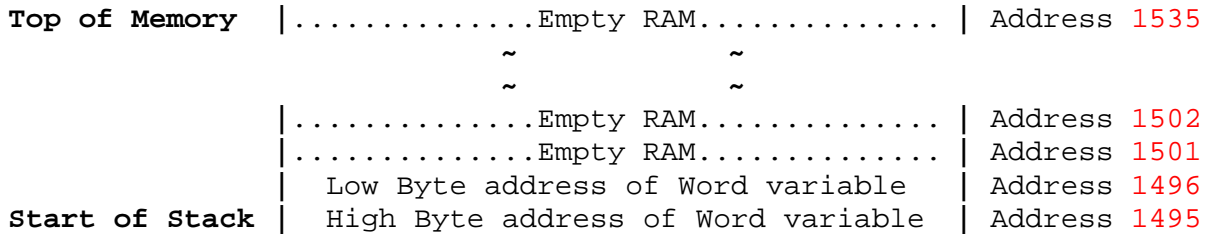
The high byte of the variable is first pushed on to the stack, then the low byte. And as you can see, the stack grows in an upward direction whenever a **Push** is implemented, which means it shrinks back down whenever a **Pop** is implemented.

If we were to **Push** a **Dword** variable on to the stack as well as the **Word** variable, the stack memory would look like: -

Top of MemoryEmpty RAM.....	Address 1535
	~	~
	~	~
Empty RAM.....	Address 1502
Empty RAM.....	Address 1501
	Low Byte address of Dword variable	Address 1500
	Mid1 Byte address of Dword variable	Address 1499
	Mid2 Byte address of Dword variable	Address 1498
	High Byte address of Dword variable	Address 1497
	Low Byte address of Word variable	Address 1496
Start of Stack	High Byte address of Word variable	Address 1495

Popping.

When using the **Pop** command, the same variable type that was pushed last must be popped first, or the stack will become out of phase and any variables that are subsequently popped will contain invalid data. For example, using the above analogy, we need to **Pop** a **Dword** variable first. The **Dword** variable will be popped Low Byte first, then MID1 Byte, then MID2 Byte, then lastly the High Byte. This will ensure that the same value pushed will be reconstructed correctly when placed into its recipient variable. After the **Pop**, the stack memory map will look like: -



If a **Word** variable was then popped, the stack will be empty, however, what if we popped a **Byte** variable instead? the stack would contain the remnants of the **Word** variable previously pushed. Now what if we popped a **Dword** variable instead of the required **Word** variable? the stack would underflow by two bytes and corrupt any variables using those address's . The compiler cannot warn you of this occurring, so it is up to you, the programmer, to ensure that proper stack management is carried out. The same is true if the stack overflows. i.e. goes beyond the top of RAM. The compiler cannot give a warning.

Technical Details of Stack implementation.

The stack implemented by the compiler is known as an **Incrementing Last-In First-Out** Stack. *Incrementing* because it grows upwards in memory. *Last-In First-Out* because the last variable pushed, will be the first variable popped.

The stack is not circular in operation, so that a stack overflow will rollover into the PICmicro's hardware register, and an underflow will simply overwrite RAM immediately below the Start of Stack memory. If a circular operating stack is required, it will need to be coded in the main BASIC program, by examination and manipulation of the stack pointer (see below).

Indirect register pair FSR2L and FSR2H are used as a 16-bit stack pointer, and are incremented for every **Byte** pushed, and decremented for every **Byte** popped. Therefore checking the FSR2 registers in the BASIC program will give an indication of the stack's condition if required. This also means that the BASIC program cannot use the FSR2 register pair as part of its code, unless for manipulating the stack. Note that none of the compiler's commands, other than **Push** and **Pop**, use FSR2.

Whenever a variable is popped from the stack, the stack's memory is not actually cleared, only the stack pointer is moved. Therefore, the above diagrams are not quite true when they show empty RAM, but unless you have use of the remnants of the variable, it should be considered as empty, and will be overwritten by the next **Push** command.

See also : **Call, Goto, Push, Pop.**

Goto

Syntax

Goto *Label*

Overview

Jump to a defined label and continue 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 Var1 = 3 Then Goto Jumpover
    {
    code here executed only if Var1<>3
    .....
    .....
    }
JumpOver:
    {continue code execution}
```

In this example, if Var1=3 then the program jumps over all the code below it until it reaches the *label* JumpOver where program execution continues as normal.

See also : **Call, Gosub.**

HbStart

Syntax HbStart

Overview

Send a **Start** condition to the I²C bus using the microcontroller's MSSP module.

Notes

Because of the subtleties involved in interfacing to some I²C devices, the compiler's standard Hbusin, and Hbusout commands were found lacking. Therefore, individual pieces of the I²C protocol may be used in association with the new structure of Hbusin, and Hbusout. See relevant sections for more information.

Example

```
' Interface to a 24LC32 serial eeprom
Device = 16F877          ' Use a device with an MSSP module
Dim Loop as Byte
Dim Array[10] as Byte
' Transmit bytes to the I2C bus
HbStart                ' Send a Start condition
Hbusout %10100000     ' Target an eeprom, and send a Write command
Hbusout 0              ' Send the HighByte of the address
Hbusout 0              ' Send the LowByte of the address
For Loop = 48 to 57   ' Create a loop containing ASCII 0 to 9
    Hbusout Loop      ' Send the value of Loop to the eeprom
Next                  ' Close the loop
HbStop                ' Send a Stop condition
DelayMs 10            ' Wait for the data to be entered into eeprom matrix
' Receive bytes from the I2C bus
HbStart                ' Send a Start condition
Hbusout %10100000     ' Target an eeprom, and send a Write command
Hbusout 0              ' Send the HighByte of the address
Hbusout 0              ' Send the LowByte of the address
HbRestart              ' Send a Restart condition
Hbusout %10100001     ' Target an eeprom, and send a Read command
For Loop = 0 to 9     ' Create a loop
    Array[Loop] = Hbusin ' Load an array with bytes received
    If Loop = 9 Then HbStop : Else : HbusAck ' Ack or Stop ?
Next                  ' Close the loop
Print At 1,1, Str Array ' Display the Array as a String
```

See also : HbusAck, HbRestart, HbStop, Hbusin, Hbusout.

HbStop

Syntax
HbStop

Overview

Send a **Stop** condition to the I²C bus using the microcontroller's MSSP module.

HbRestart

Syntax
HbRestart

Overview

Send a **Restart** condition to the I²C bus using the microcontroller's MSSP module.

HbusAck

Syntax
HbusAck

Overview

Send an **Acknowledge** condition to the I²C bus using the microcontroller's MSSP module.

HbusNack

Syntax
HbusNack

Overview

Send a **Not Acknowledge** condition to the I²C bus using the microcontroller's MSSP module..

See also : **HbStart, HbRestart, HbStop, Hbusin, Hbusout.**

Hbusin

Syntax

Variable = **Hbusin** *Control*, { *Address* }

or

Variable = **Hbusin**

or

Hbusin *Control*, { *Address* }, [*Variable* {, *Variable*...}]

or

Hbusin *Variable*

Overview

Receives a value from the I²C bus using the MSSP module, and places it into *variable/s*. If variations *two* or *four* (see above) are used, then No Acknowledge, or Stop is sent after the data. Variations *one* and *three* first send the *control* and optional *address*.

Operators

Variable is a user defined variable or constant.

Control may be a constant value or a **Byte** sized variable expression.

Address may be a constant value or a variable expression.

The four variations of the **Hbusin** command may be used in the same BASIC program. The *second* and *fourth* types (see above) are useful for simply receiving a single byte from the bus, and must be used in conjunction with one of the low level commands. i.e. **HbStart**, **HbRestart**, **HbusAck**, or **HbStop**. The *first*, and *third* types may be used to receive several values and designate each to a separate variable, or variable type.

The **Hbusin** command operates as an I²C master, using the microcontroller's MSSP module, and may be used to interface with any device that complies with the 2-wire I²C protocol.

The most significant 7-bits of *control* byte contain the control code and the slave address of the device being interfaced with. Bit-0 is the flag that indicates whether a read or write command is being implemented.

For example, if we were interfacing to an external eeprom such as the 24LC32, the control code would be %10100001 or \$A1. The most significant 4-bits (1010) are the eeprom's unique slave address. Bits 2 to 3 reflect the three address pins of the eeprom. And bit-0 is set to signify that we wish to read from the eeprom. Note that this bit is automatically set by the **Hbusin** command, regardless of its initial setting.

Example

' Receive a byte from the I2C bus and place it into variable Var1.

```
Dim Var1 as Byte           ' We'll only read 8-bits
Dim Address as Word       ' 16-bit address required
Symbol Control %10100001  ' Target an eeprom
Address = 20              ' Read the value at address 20
Var1 = Hbusin Control, Address ' Read the byte from the eeprom
```

or

```
Hbusin Control, Address, [Var1] ' Read the byte from the eeprom
```

Address, is an optional parameter that may be an 8-bit or 16-bit value. If a variable is used in this position, the size of *address* is dictated by the size of the variable used (**Byte** or **Word**). In the case of the previous eeprom interfacing, the 24LC32 eeprom requires a 16-bit address. While the smaller types require an 8-bit address. Make sure you assign the right size address for the device interfaced with, or you may not achieve the results you intended.

The value received from the bus depends on the size of the variables used, except for variation three, which only receives a **Byte** (8-bits). For example: -

```
Dim Wrđ as Word           ' Declare a Word size variable
Wrđ = Hbusin Control, Address
```

Will receive a 16-bit value from the bus. While: -

```
Dim Var1 as Byte         ' Declare a Byte size variable
Var1 = Hbusin Control, Address
```

Will receive an 8-bit value from the bus.

Using the *third* variation of the **Hbusin** command allows differing variable assignments. For example: -

```
Dim Var1 as Byte
Dim Wrđ as Word
Hbusin Control, Address, [Var1, Wrđ]
```

Will receive two values from the bus, the first being an 8-bit value dictated by the size of variable Var1 which has been declared as a byte. And a 16-bit value, this time dictated by the size of the variable Wrđ which has been declared as a word. Of course, bit type variables may also be used, but in most cases these are not of any practical use as they still take up a byte within the eeprom.

The *second* and *fourth* variations allow all the subtleties of the I²C protocol to be exploited, as each operation may be broken down into its constituent parts. It is advisable to refer to the datasheet of the device being interfaced to fully understand its requirements. See section on **HbStart**, **HbRestart**, **HbusAck**, or **HbStop**, for example code.

Hbusin Declare

```
Declare Hbus_Bitrate Constant 100, 400, 1000
```

The standard speed for the I²C bus is 100KHz. Some devices use a higher bus speed of 400KHz. The above Declare allows the I²C bus speed to be increased or decreased. Use this Declare with caution, as too high a bit rate may exceed the device's specs, which will result in intermittent transactions, or in some cases, no transactions at all. The datasheet for the device used will inform you of its bus speed. The default bit rate is the standard 100KHz.

Notes

Not all PICmicro™ devices contain an MSSP module, some only contain an SSP type, which only allows I²C slave operations. These types of devices may not be used with any of the HBUS commands. Therefore, always read and understand the datasheet for the PICmicro™ device used.

When the **Hbusin** command is used, the appropriate SDA and SCL Port and Pin are automatically setup as inputs. The SDA, and SCL lines are predetermined as hardware pins on the PICmicro™ i.e. For a 16F877 device, the SCL pin is PORTC.3, and SDA is PORTC.4. Therefore, there is no need to pre-declare these.

Because the I²C protocol calls for an *open-collector* interface, pull-up resistors are required on both the SDA and SCL lines. Values of 1KΩ to 4.7KΩ will suffice.

Str modifier with Hbusin

Using the **Str** modifier allows variations *three* and *four* of the **Hbusin** command to transfer the bytes received from the I²C bus directly into a byte array. If the amount of received characters is not enough to fill the entire array, then a formatter may be placed after the array's name, which will only receive characters until the specified length is reached. An example of each is shown below: -

```
Dim Array[10] as Byte ' Define an array of 10 bytes
Dim Address as Byte ' Create a word sized variable

Hbusin %10100000, Address, [Str Array] ' Load data into all the array
' Load data into only the first 5 elements of the array
Hbusin %10100000, Address, [Str Array\5]
HbStart ' Send a Start condition
Hbusout %10100000 ' Target an eeprom, and send a WRITE command
Hbusout 0 ' Send the HighByte of the address
Hbusout 0 ' Send the LowByte of the address
HbRestart ' Send a Restart condition
Hbusout %10100001 ' Target an eeprom, and send a Read command
Hbusin Str Array ' Load all the array with bytes received
HbStop ' Send a Stop condition
```

An alternative ending to the above example is: -

```
Hbusin Str Array\5 ' Load data into only the first 5 elements of the array
HbStop ' Send a Stop condition
```

See also : HbusAck, HbRestart, HbStop, HbStart, Hbusout.

Hbusout

Syntax

Hbusout *Control*, { *Address* }, [*Variable* {, *Variable...*}]

or

Hbusout *Variable*

Overview

Transmit a value to the I²C bus using the microcontroller's on-board MSSP module, by first sending the *control* and optional *address* out of the clock pin (*SCL*), and data pin (*SDA*). Or alternatively, if only one operator is included after the **Hbusout** command, a single value will be transmitted, along with an Ack reception.

Operators

Variable is a user defined variable or constant.

Control may be a constant value or a **Byte** sized variable expression.

Address may be a constant, variable, or expression.

The **Hbusout** command operates as an I²C master and may be used to interface with any device that complies with the 2-wire I²C protocol.

The most significant 7-bits of *control* byte contain the control code and the slave address of the device being interfaced with. Bit-0 is the flag that indicates whether a read or write command is being implemented.

For example, if we were interfacing to an external eeprom such as the 24LC32, the control code would be %10100000 or \$A0. The most significant 4-bits (1010) are the eeprom's unique slave address. Bits 2 to 3 reflect the three address pins of the eeprom. And Bit-0 is clear to signify that we wish to write to the eeprom. Note that this bit is automatically cleared by the **Hbusout** command, regardless of its initial value.

Example

' Send a byte to the I2C bus.

```
Dim Var1 as Byte           ' We'll only read 8-bits
Dim Address as Word       ' 16-bit address required
Symbol Control = %10100000 ' Target an eeprom
Address = 20               ' Write to address 20
Var1 = 200                 ' The value place into address 20
Hbusout Control, Address, [Var1] ' Send the byte to the eeprom
DelayMs 10                 ' Allow time for allocation of byte
```

Address, is an optional parameter that may be an 8-bit or 16-bit value. If a variable is used in this position, the size of *address* is dictated by the size of the variable used (**Byte** or **Word**). In the case of the above eeprom interfacing, the 24LC32 eeprom requires a 16-bit address. While the smaller types require an 8-bit address. Make sure you assign the right size address for the device interfaced with, or you may not achieve the results you intended.

The value sent to the bus depends on the size of the variables used. For example: -

```
Dim Wrd as Word           ' Declare a Word size variable
Hbusout Control, Address, [Wrd]
```

Will send a 16-bit value to the bus. While: -

```
Dim Var1 as Byte         ' Declare a Byte size variable
Hbusout Control, Address, [Var1]
```

Will send an 8-bit value to the bus.

Using more than one variable within the brackets allows differing variable sizes to be sent. For example: -

```
Dim Var1 as Byte
Dim Wrd as Word
Hbusout Control, Address, [Var1, Wrd]
```

Will send two values to the bus, the first being an 8-bit value dictated by the size of variable Var1 which has been declared as a byte. And a 16-bit value, this time dictated by the size of the variable Wrd which has been declared as a word. Of course, **Bit** type variables may also be used, but in most cases these are not of any practical use as they still take up a byte within the eeprom.

A string of characters can also be transmitted, by enclosing them in quotes: -

```
Hbusout Control, Address, [ "Hello World", Var1, Wrd ]
```

Using the second variation of the **Hbusout** command, necessitates using the low level commands i.e. **HbStart**, **HbRestart**, **HbusAck**, or **HbStop**.

Using the **Hbusout** command with only one value after it, sends a byte of data to the I²C bus, and returns holding the Acknowledge reception. This acknowledge indicates whether the data has been received by the slave device.

The Ack reception is returned in the PICmicro's CARRY flag, which is STATUS.0, and also System variable PP4.0. A value of zero indicates that the data was received correctly, while a one indicates that the data was not received, or that the slave device has sent a NAck return. You must read and understand the datasheet for the device being interfacing to, before the Ack return can be used successfully. An code snippet is shown below: -

```
' Transmit a byte to a 24LC32 serial eeprom
Dim PP4 as Byte System
HbStart           ' Send a Start condition
Hbusout %10100000 ' Target an eeprom, and send a Write command
Hbusout 0         ' Send the HighByte of the address
Hbusout 0         ' Send the LowByte of the address
Hbusout "A"      ' Send the value 65 to the bus
If PP4.0 = 1 Then Goto Not_Received ' Has Ack been received OK ?
HbStop           ' Send a Stop condition
DelayMs 10       ' Wait for the data to be entered into eeprom matrix
```


Str modifier with Hbusout.

The **Str** modifier is used for transmitting a string of bytes from a byte array variable. A string is a set of bytes sized values that are arranged or accessed in a certain order. The values 1, 2, 3 would be stored in a string with the value 1 first, followed by 2 then followed by the value 3. A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string 1,2,3 would be stored in a byte array containing three bytes (elements).

Below is an example that sends four bytes from an array: -

```
Dim MyArray[10] as Byte ' Create a 10-byte array.
MyArray [0] = "A"      ' Load the first 4 bytes of the array
MyArray [1] = "B"      ' With the data to send
MyArray [2] = "C"
MyArray [3] = "D"
Hbusout %10100000, Address, [Str MyArray \4] ' Send 4-byte string.
```

Note that we use the optional `\n` argument of **Str**. If we didn't specify this, the program would try to keep sending characters until all 10 bytes of the array were transmitted. Since we do not wish all 10 bytes to be transmitted, we chose to tell it explicitly to only send the first 4 bytes.

The above example may also be written as: -

```
Dim MyArray [10] as Byte ' Create a 10-byte array.
Str MyArray = "ABCD"     ' Load the first 4 bytes of the array
HbStart                 ' Send a Start condition
Hbusout %10100000       ' Target an eeprom, and send a Write command
Hbusout 0                ' Send the HighByte of the address
Hbusout 0                ' Send the LowByte of the address
Hbusout Str MyArray\4   ' Send 4-byte string.
HbStop                  ' Send a Stop condition
```

The above example, has exactly the same function as the previous one. The only differences are that the string is now constructed using the **Str** as a command instead of a modifier, and the low-level Hbus commands have been used.

Notes

Not all PICmicro™ devices contain an MSSP module, some only contain an SSP type, which only allows I²C slave operations. These types of devices may not be used with any of the Hbus commands. Therefore, always read and understand the datasheet for the PICmicro™ device used.

When the **Hbusout** command is used, the appropriate SDA and SCL Port and Pin are automatically setup as inputs. The SDA, and SCL lines are predetermined as hardware pins on the PICmicro™ i.e. For a 16F877 device, the SCL pin is PORTC.3, and SDA is PORTC.4. Therefore, there is no need to pre-declare these. Because the I²C protocol calls for an *open-collector* interface, pull-up resistors are required on both the SDA and SCL lines. Values of 1KΩ to 4.7KΩ will suffice.

See also : HbusAck, HbRestart, HbStop, Hbusin, HbStart.

High

Syntax

High *Port* or *Port.Bit*

Overview

Place a Port or bit in a high state. For a Port, this means filling it with 1's. For a bit this means setting it to 1.

Operators

Port can be any valid port.

Port.Bit can be any valid port and bit combination, i.e. PORTA.1

Example

```
Symbol LED = PORTB.4
```

```
High LED
```

See also : Clear, Dim, Low, Set, Symbol.

Hpwm

Syntax

Hpwm Channel, DutyCycle, Frequency

Overview

Output a pulse width modulated pulse train using the CCP modules Pwm hardware, available on some PICmicros. The Pwm pulses produced can run continuously in the background while the program is executing other instructions.

Operators

Channel is a constant value that specifies which hardware Pwm channel to use. Some devices have 1, 2 or 3 Pwm channels. On devices with 2 channels, the Frequency must be the same on both channels. It must be noted, that this is a limitation of the PICmicro™ not the compiler. The data sheet for the particular device used shows the fixed hardware pin for each Channel. For example, for a PIC16F877, Channel 1 is CCP1 which is pin PORTC.2. Channel 2 is CCP2 which is pin PORTC.1.

DutyCycle is a variable, constant (0-255), or expression that specifies the on/off (high/low) ratio of the signal. It ranges from 0 to 255, where 0 is off (low all the time) and 255 is on (high) all the time. A value of 127 gives a 50% duty cycle (square wave).

Frequency is a variable, constant (0-32767), or expression that specifies the desired frequency of the Pwm signal. Not all frequencies are available at all oscillator settings. The highest frequency at any oscillator speed is 32767Hz. The lowest usable **Hpwm Frequency** at each oscillator setting is shown in the table below: -

Xtal frequency	Lowest useable Pwm frequency
4MHz	145Hz
8MHz	489Hz
10MHz	611Hz
12MHz	733Hz
16MHz	977Hz
20MHz	1221Hz
24MHz	1465Hz
33MHz	2015Hz
40MHz	2442Hz

Example

```
Device = 16F877
Declare Xtal = 20
Hpwm 1,127,1000      ' Send a 50% duty cycle Pwm signal at 1KHz
DelayMs 500
Hpwm 1,64,2000      ' Send a 25% duty cycle Pwm signal at 2KHz
Stop
```

Notes

Some devices, such as the PIC16F62x, and PIC18F4xx, have alternate pins that may be used for **Hpwm**. The following **Declares** allow the use of different pins: -

```
Declare CCP1_Pin Port . Pin      ' Select Hpwm port and bit for CCP1 module.
Declare CCP2_Pin Port . Pin      ' Select Hpwm port and bit for CCP2 module.
```

See also : Pwm, Pulseout, Servo.

Hrsin

Syntax

Variable = **Hrsin**, { *Timeout*, *Timeout Label* }

or

Hrsin { *Timeout*, *Timeout Label* }, { *Parity Error Label* }, *Modifiers*, *Variable* {, *Variable...* }

Overview

Receive one or more values from the serial port on devices that contain a hardware USART.

Operators

Timeout is an *optional* value for the length of time the **Hrsin** command will wait before jumping to label **Timeout Label**. **Timeout** is specified in 1 millisecond units and has a maximum of 16-bits.

Timeout Label is an *optional* valid BASIC label where **Hrsin** will jump to in the event that a character has not been received within the time specified by **Timeout**.

Parity Error Label is an *optional* valid BASIC label where **Hrsin** will jump to in the event that a Parity error is received. Parity is set using **Declares**. Parity Error detecting is not supported in the inline version of **Hrsin** (first syntax example above).

Modifier is one of the many formatting modifiers, explained below.

Variable is a **Bit**, **Byte**, **Word**, or **Dword** variable, that will be loaded by **Hrsin**.

Example

```
' Receive values serially and timeout if no reception after 1 second
Device 16F877
Declare Xtal = 4

Declare Hserial_Baud = 9600           ' Set baud rate to 9600
Declare Hserial_RCSTA = %10010000    ' Enable continuous receive
Declare Hserial_TXSTA = %00100000    ' Enable transmit and asynchronous mode
Declare Hserial_Clear = On           ' Clear the buffer before receiving
Dim Var1 as Byte

Loop:
  Var1 = Hrsin, {1000, Timeout}      ' Receive a byte serially into Var1
  Print Dec Var1, " "                ' Display the byte received
  Goto Loop                          ' Loop forever
Timeout:
  Cls
  Print "TIMED OUT"                  ' Display an error if Hrsin timed out
  Stop
```

Hrsin Modifiers.

As we already know, **Rsin** will wait for and receive a single byte of data, and store it in a variable . If the PICmicro™ were connected to a PC running a terminal program and the user pressed the "A" key on the keyboard, after the **Hrsin** command executed, the variable would contain 65, which is the ASCII code for the letter "A"

What would happen if the user pressed the "1" key? The result would be that the variable would contain the value 49 (the ASCII code for the character "1"). This is an important point to remember: every time you press a character on the keyboard, the computer receives the ASCII value of that character. It is up to the receiving side to interpret the values as necessary.

In this case, perhaps we actually wanted the variable to end up with the value 1, rather than the ASCII code 49.

The **Hrsin** command provides a modifier, called the decimal modifier, which will interpret this for us. Look at the following code: -

```
Dim SerData as Byte
Hrsin Dec SerData
```

Notice the decimal modifier in the **Hrsin** command that appears just to the left of the SerData variable. This tells **Hrsin** to convert incoming text representing decimal numbers into true decimal form and store the result in SerData. If the user running the terminal software pressed the "1", "2" and then "3" keys followed by a space or other non-numeric text, the value 123 will be stored in the variable SerData, allowing the rest of the program to perform any numeric operation on the variable.

Without the decimal modifier, however, you would have been forced to receive each character ("1", "2" and "3") separately, and then would still have to do some manual conversion to arrive at the number 123 (one hundred twenty three) before you can do the desired calculations on it.

The decimal modifier is designed to seek out text that represents decimal numbers. The characters that represent decimal numbers are the characters "0" through "9". Once the **Hrsin** command is asked to use the decimal modifier for a particular variable, it monitors the incoming serial data, looking for the first decimal character. Once it finds the first decimal character, it will continue looking for more (accumulating the entire multi-digit number) until it finds a non-decimal numeric character. Remember that it will not finish until it finds at least one decimal character followed by at least one non-decimal character.

To illustrate this further, examine the following examples (assuming we're using the same code example as above): -

Serial input: "ABC"

Result: The program halts at the **Hrsin** command, continuously waiting for decimal text.

Serial input: "123" (with no characters following it)

Result: The program halts at the **Hrsin** command. It recognises the characters "1", "2" and "3" as the number one hundred twenty three, but since no characters follow the "3", it waits continuously, since there's no way to tell whether 123 is the entire number or not.

Serial input: "123" (followed by a space character)

Result: Similar to the above example, except once the space character is received, the program knows the entire number is 123, and stores this value in SerData. The **Hrsin** command then ends, allowing the next line of code to run.

Serial input: "123A"

Result: Same as the example above. The "A" character, just like the space character, is the first non-decimal text after the number 123, indicating to the program that it has received the entire number.

Serial input: "ABCD123EFGH"

Result: Similar to examples 3 and 4 above. The characters "ABCD" are ignored (since they're not decimal text), the characters "123" are evaluated to be the number 123 and the following character, "E", indicates to the program that it has received the entire number.

The final result of the **Dec** modifier is limited to 16 bits (up to the value 65535). If a value larger than this is received by the decimal modifier, the end result will be incorrect because the result rolled-over the maximum 16-bit value. Therefore, **Hrsin** modifiers may not (at this time) be used to load **Dword** (32-bit) variables.

The decimal modifier is only one of a family of conversion modifiers available with **Hrsin**. See below for a list of available conversion modifiers. All of the conversion modifiers work similar to the decimal modifier (as described above). The modifiers receive bytes of data, waiting for the first byte that falls within the range of characters they accept (e.g., "0" or "1" for binary, "0" to "9" for decimal, "0" to "9" and "A" to "F" for hex). Once they receive a numeric character, they keep accepting input until a non-numeric character arrives, or in the case of the fixed length modifiers, the maximum specified number of digits arrives.

While very effective at filtering and converting input text, the modifiers aren't completely fool-proof. As mentioned before, many conversion modifiers will keep accepting text until the first non-numeric text arrives, even if the resulting value exceeds the size of the variable. After **Hrsin**, a **Byte** variable will contain the lowest 8 bits of the value entered and a **Word** (16-bits) would contain the lowest 16 bits. You can control this to some degree by using a modifier that specifies the number of digits, such as **Dec2**, which would accept values only in the range of 0 to 99.

Conversion Modifier	Type of Number	Numeric	Characters Accepted
Dec {1..10}	Decimal, optionally limited	to 1 - 10 digits	0 through 9
Hex {1..8}	Hexadecimal, optionally limited	to 1 - 8 digits	0 through 9, A through F
Bin {1..32}	Binary, optionally limited	to 1 - 32 digits	0, 1

A variable preceded by **Bin** will receive the ASCII representation of its binary value. For example, if **Bin** Var1 is specified and "1000" is received, Var1 will be set to 8.

A variable preceded by **Dec** will receive the ASCII representation of its decimal value. For example, if **Dec** Var1 is specified and "123" is received, Var1 will be set to 123.

A variable preceded by **Hex** will receive the ASCII representation of its hexadecimal value. For example, if **Hex** Var1 is specified and "FE" is received, Var1 will be set to 254.

SKIP followed by a count will skip that many characters in the input stream. For example, **SKIP** 4 will skip 4 characters.

The **Hrsin** command can be configured to wait for a specified sequence of characters before it retrieves any additional input. For example, suppose a device attached to the PICmicro™ is known to send many different sequences of data, but the only data you wish to observe happens to appear right after the unique characters, "XYZ". A modifier named **Wait** can be used for this purpose: -

```
Hrsin Wait("XYZ"), SerData
```

The above code waits for the characters "X", "Y" and "Z" to be received, in that order, then it receives the next data byte and places it into variable SerData.

Str modifier.

The **Hrsin** command also has a modifier for handling a string of characters, named **Str**.

The **Str** modifier is used for receiving a string of characters into a byte array variable.

A string is a set of characters that are arranged or accessed in a certain order. The characters "ABC" would be stored in a string with the "A" first, followed by the "B" then followed by the "C". A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string "ABC" would be stored in a byte array containing three bytes (elements).

Below is an example that receives ten bytes and stores them in the 10-byte array, SERString: -

```
Dim SerString[10] as Byte      ' Create a 10-byte array.
Hrsin Str SerString           ' Fill the array with received data.
Print Str SerString           ' Display the string.
```

If the amount of received characters is not enough to fill the entire array, then a formatter may be placed after the array's name, which will only receive characters until the specified length is reached. For example: -

```
Dim SerString[10] as Byte      ' Create a 10-byte array.
Hrsin Str SerString\5         ' Fill the first 5-bytes of the array
Print Str SerString\5         ' Display the 5-character string.
```

The example above illustrates how to fill only the first n bytes of an array, and then how to display only the first n bytes of the array. n refers to the value placed after the backslash.

Because of its complexity, serial communication can be rather difficult to work with at times. Using the guidelines below when developing a project using the **Hrsin** and **Hrsout** commands may help to eliminate some obvious errors: -

Always build your project in steps.

Start with small, manageable pieces of code, (that deal with serial communication) and test them, one individually.

Add more and more small pieces, testing them each time, as you go.

Never write a large portion of code that works with serial communication without testing its smallest workable pieces first.

Pay attention to timing.

Be careful to calculate and overestimate the amount of time, operations should take within the PICmicro™ for a given oscillator frequency. Misunderstanding the timing constraints is the source of most problems with code that communicate serially. If the serial communication in your project is bi-directional, the above statement is even more critical.

Pay attention to wiring.

Take extra time to study and verify serial communication wiring diagrams. A mistake in wiring can cause strange problems in communication, or no communication at all. Make sure to connect the ground pins (Vss) between the devices that are communicating serially.

Verify port setting on the PC and in the Hrsin / Hrsout commands.

Unmatched settings on the sender and receiver side will cause garbled data transfers or no data transfers. This is never more critical than when a line transceiver is used (i.e. MAX232). Always remember that a line transceiver inverts the serial polarity.

If the serial data received is unreadable, it is most likely caused by a baud rate setting error, or a polarity error.

If receiving data from another device that is not a PICmicro™, try to use baud rates of 9600 and below, or alternatively, use a higher frequency crystal.

Because of additional overheads in the PICmicro™, and the fact that the **Hrsin** command only offers a 2 level receive buffer for serial communication, received data may sometimes be missed or garbled. If this occurs, try lowering the baud rate, or increasing the crystal frequency. Using simple variables (not arrays) will also increase the chance that the PICmicro™ will receive the data properly.

Declares

There are five Declare directives for use with **Hrsin**. These are: -

Declare Hserial_Baud Constant value

Sets the BAUD rate that will be used to receive a value serially. The baud rate is calculated using the **Xtal** frequency declared in the program. The default baud rate if the Declare is not included in the program listing is 2400 baud.

Declare Hserial_RCSTA Constant value (0 to 255)

Hserial_RCSTA, sets the respective PICmicro™ hardware register RCSTA, to the value in the Declare. See the Microchip data sheet for the device used for more information regarding this register.

Declare Hserial_TXSTA Constant value (0 to 255)

Hserial_TXSTA, sets the respective PICmicro™ hardware register, TXSTA, to the value in the Declare. See the Microchip data sheet for the device used for more information regarding this register. The TXSTA register BRGH bit (bit 2) controls the high speed mode for the baud rate generator. Certain baud rates at certain oscillator speeds require this bit to be set to operate properly. To do this, set **Hserial_TXSTA** to a value of 24h instead of the normal 20h. Refer to the Microchip data sheet for the hardware serial port baud rate tables and additional information.

Declare Hserial_Parity Odd or Even

Enables/Disables parity on the serial port. For both **Hrsin** and **Hrsout** The default serial data format is 8N1, 8 data bits, no parity bit and 1 stop bit. 7E1 (7 data bits, even parity, 1 stop bit) or 7O1 (7 data bits, odd parity, 1 stop bit) may be enabled using the **Hserial_Parity** declare.

```
Declare Hserial_Parity = Even      ' Use if even parity desired
```

```
Declare Hserial_Parity = Odd       ' Use if odd parity desired
```

Declare Hserial_Clear On or Off

Clear the overflow error bit before commencing a read.

Because the hardware serial port only has a 2-byte input buffer, it can easily overflow if characters are not read from it often enough. When this occurs, the USART stops accepting any new

characters, and requires resetting. This overflow error can be reset by strobing the CREN bit within the RCSTA register. Example: -

```
RCSTA.4 = 0
RCSTA.4 = 1
```

OR

```
Clear RCSTA.4
Set RCSTA.4
```

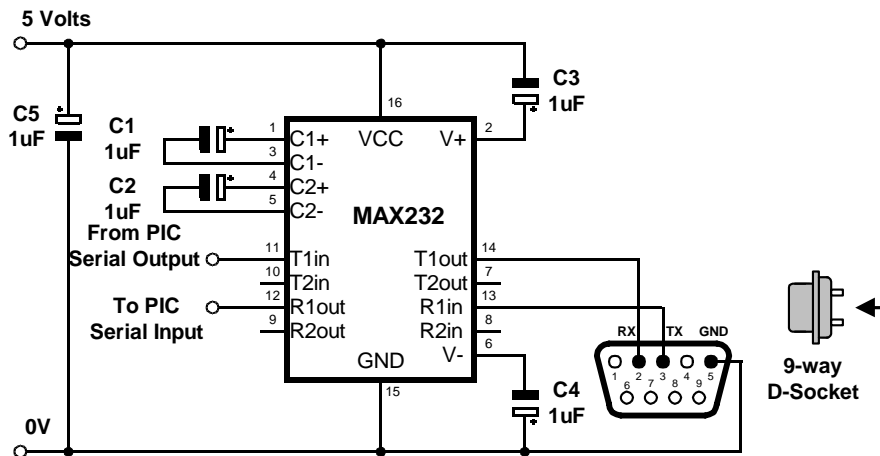
Alternatively, the **Hserial_Clear** declare can be used to automatically clear this error, even if no error occurred. However, the program will not know if an error occurred while reading, therefore some characters may be lost.

```
Declare Hserial_Clear = On
```

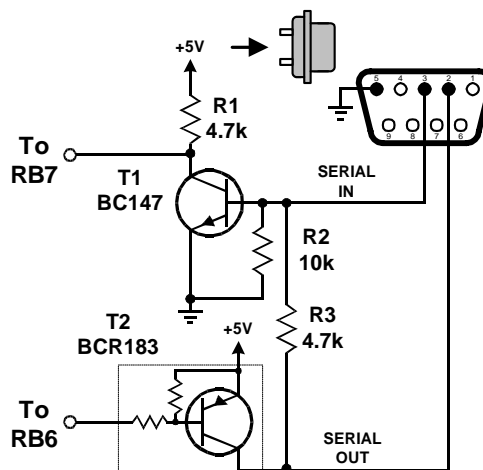
Notes

Hrsin can only be used with devices that contain a hardware USART. See the specific device's data sheet for further information concerning the serial input pin as well as other relevant parameters.

Since the serial transmission is done in hardware, it is not possible to set the levels to an inverted state to eliminate an RS232 driver. Therefore a suitable driver should be used with **Hrsin**. Just such a circuit using a MAX232 is shown below.



A simpler, and somewhat more elegant transceiver circuit using only 5 discrete components is shown in the diagram below.



See also :Declare, Rsin, Rsout, Serin, Serout, Hrsout, Hserin, Hserout.

Hrsout

Syntax

Hrsout *Item* {, *Item...* }

Overview

Transmit one or more *Items* from the hardware serial port on devices that support asynchronous serial communications in hardware.

Operators

Item may be a constant, variable, expression, string list, or inline command.

There are no operators as such, instead there are *modifiers*. For example, if an at sign '@' precedes an *Item*, the ASCII representation for each digit is transmitted.

The modifiers are listed below: -

Modifier	Operation
At ypos,xpos	Position the cursor on a serial LCD
ClS	Clear a serial LCD (also creates a 30ms delay)
Bin{1..32}	Send binary digits
Dec{1..10}	Send decimal digits
Hex{1..8}	Send hexadecimal digits
Sbin{1..32}	Send signed binary digits
Sdec{1..10}	Send signed decimal digits
Shex{1..8}	Send signed hexadecimal digits
Ibin{1..32}	Send binary digits with a preceding '%' identifier
Idec{1..10}	Send decimal digits with a preceding '#' identifier
Ihex{1..8}	Send hexadecimal digits with a preceding '\$' identifier
ISbin{1..32}	Send signed binary digits with a preceding '%' identifier
ISdec{1..10}	Send signed decimal digits with a preceding '#' identifier
IShex{1..8}	Send signed hexadecimal digits with a preceding '\$' identifier
Rep c\n	Send character c repeated n times
Str array\n	Send all or part of an array
Cstr cdata	Send string data defined in a Cdata statement.

The numbers after the **Bin**, **Dec**, and **Hex** modifiers are optional. If they are omitted, then the default is all the digits that make up the value will be displayed.

If a floating point variable is to be displayed, then the digits after the **Dec** modifier determine how many remainder digits are send. i.e. numbers after the decimal point.

```
Dim Flt as Float
Flt = 3.145
Hrsout Dec2 Flt ' Send 2 digits after the decimal point
```

The above program will transmit the ASCII characters "3.14"

If the digit after the **Dec** modifier is omitted, then 3 digits will be displayed after the decimal point.

```
Dim Flt as Float
Flt = 3.1456
Hrsout Dec Flt ' Send 3 digits after the decimal point
```

The above program will transmit the ASCII characters "3.145"

There is no need to use the **Sdec** modifier for signed floating point values, as the compiler's **Dec** modifier will automatically display a minus result: -

```
Dim Flt as Float
Flt = -3.1456
Hrsout Dec Flt ' Send 3 digits after the decimal point
```

The above program will transmit the ASCII characters "-3.145"

Hex or **Bin** modifiers cannot be used with floating point values or variables.

The Xpos and Ypos values in the **At** modifier both start at 1. For example, to place the text "HELLO WORLD" on line 1, position 1, the code would be: -

```
Hrsout At 1, 1, "HELLO WORLD"
```

Example 1

```
Dim Var1 as Byte
Dim Wrđ as Word
Dim Dwd as Dword

Hrsout "Hello World" ' Display the text "Hello World"
Hrsout "Var1= ", Dec Var1 ' Display the decimal value of Var1
Hrsout "Var1= ", Hex Var1 ' Display the hexadecimal value of Var1
Hrsout "Var1= ", Bin Var1 ' Display the binary value of Var1
Hrsout "Dwd= ", Hex6 Dwd ' Display 6 hex characters of a Dword variable
```

Example 2

```
' Display a negative value on a serial LCD.
Symbol Negative = -200
Hrsout At 1, 1, Sdec Negative
```

Example 3

```
' Display a negative value on a serial LCD with a preceding identifier.
Hrsout At 1, 1, IShex -$1234
```

Example 3 will produce the text "\$-1234" on the LCD.

Some microcontrollers such as the 16F87x, and 18FXXX range have the ability to read and write to their own flash memory. And although writing to this memory too many times is unhealthy for the PICmicro™, reading this memory is both fast, and harmless. Which offers a unique form of data storage and retrieval, the **Cdata** command proves this, as it uses the mechanism of reading and storing in the device's flash memory.

Combining the unique features of the 'self modifying PICmicro's' with a string format, the compiler is capable of reducing the overhead of printing, or transmitting large amounts of text data.

The **Cstr** modifier may be used in commands that deal with text processing i.e. **Serout**, **Hserout**, and **Print** etc.

The **Cstr** modifier is used in conjunction with the **Cdata** command. The **Cdata** command is used for initially creating the string of characters: -

```
String1: Cdata "HELLO WORLD", 0
```

The above line of code will create, in flash memory, the values that make up the ASCII text "HELLO WORLD", at address String1. Note the null terminator after the ASCII text.

null terminated means that a zero (null) is placed at the end of the string of ASCII characters to signal that the string has finished.

To display, or transmit this string of characters, the following command structure could be used:

```
Hrsout Cstr String1
```

The label that declared the address where the list of **Cdata** values resided, now becomes the string's name. In a large program with lots of text formatting, this type of structure can save quite literally hundreds of bytes of valuable code space.

Try both these small programs, and you'll see that using **Cstr** saves a few bytes of code: -

First the standard way of displaying text: -

```
Device = 16F877
Cls
Hrsout "HELLO WORLD",13
Hrsout "HOW ARE YOU?",13
Hrsout "I AM FINE!",13
Stop
```

Now using the **Cstr** modifier: -

```
Cls
Hrsout Cstr TEXT1
Hrsout Cstr TEXT2
Hrsout Cstr TEXT3
Stop
```

```
TEXT1: Cdata "HELLO WORLD", 13, 0
TEXT2: Cdata "HOW ARE YOU?", 13, 0
TEXT3: Cdata "I AM FINE!", 13, 0
```

Again, note the null terminators after the ASCII text in the **Cdata** commands. Without these, the PICmicro™ will continue to transmit data in an endless loop.

The term 'virtual string' relates to the fact that a string formed from the **Cdata** command cannot be written too, but only read from.

The **Str** modifier is used for sending a string of bytes from a byte array variable. A string is a set of bytes sized values that are arranged or accessed in a certain order. The values 1, 2, 3 would be stored in a string with the value 1 first, followed by 2 then followed by the value 3. A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string 1,2,3 would be stored in a byte array containing three bytes (elements).

Below is an example that displays four bytes (from a byte array): -

```
Dim MyArray[10] as Byte      ' Create a 10-byte array.
MyArray [0] = "H"           ' Load the first 5 bytes of the array
MyArray [1] = "E"           ' With the data to send
MyArray [2] = "L"
MyArray [3] = "L"
MyArray [4] = "O"
Hrsout Str MyArray\5        ' Display a 5-byte string.
```

Note that we use the optional \n argument of **Str**. If we didn't specify this, the PICmicro™ would try to keep sending characters until all 10 bytes of the array were transmitted. Since we do not wish all 10 bytes to be transmitted, we chose to tell it explicitly to only send the first 5 bytes.

The above example may also be written as: -

```
Dim MyArray [10] as Byte    ' Create a 10-byte array.
Str MyArray = "HELLO"       ' Load the first 5 bytes of the array
Hrsout Str MyArray\5        ' Send 5-byte string.
```

The above example, has exactly the same function as the previous one. The only difference is that the string is now constructed using **Str** as a command instead of a modifier.

Declares

There are four **Declare** directives for use with **Hrsout**. These are: -

Declare Hserial_Baud Constant value

Sets the BAUD rate that will be used to transmit a value serially. The baud rate is calculated using the **Xtal** frequency declared in the program. The default baud rate if the Declare is not included in the program listing is 2400 baud.

Declare Hserial_RCSTA Constant value (0 to 255)

Hserial_RCSTA, sets the respective PICmicro™ hardware register RCSTA, to the value in the Declare. See the Microchip data sheet for the device used for more information regarding this register. Refer to the upgrade manual pages for a description of the RCSTA register.

Declare Hserial_TXSTA Constant value (0 to 255)

Hserial_TXSTA, sets the respective PICmicro™ hardware register, TXSTA, to the value in the Declare. See the Microchip data sheet for the device used for more information regarding this register. The TXSTA register BRGH bit (bit 2) controls the high speed mode for the baud rate generator. Certain baud rates at certain oscillator speeds require this bit to be set to operate properly. To do this, set **Hserial_TXSTA** to a value of 24h instead of the normal 20h. Refer to the Microchip data sheet for the hardware serial port baud rate tables and additional information. Refer to the upgrade manual pages for a description of the TXSTA register.

Declare Hserial_Parity Odd or Even

Enables/Disables parity on the serial port. For both **Hrsout** and **Hrsin** The default serial data format is 8N1, 8 data bits, no parity bit and 1 stop bit. 7E1 (7 data bits, even parity, 1 stop bit) or 7O1 (7 data bits, odd parity, 1 stop bit) may be enabled using the **Hserial_Parity** declare.

```
Declare Hserial_Parity = Even      ' Use if even parity desired
Declare Hserial_Parity = Odd      ' Use if odd parity desired
```

Notes

Hrsout can only be used with devices that contain a hardware USART. See the specific device's data sheet for further information concerning the serial input pin as well as other relevant parameters.

Since the serial transmission is done in hardware, it is not possible to set the levels to an inverted state in order to eliminate an RS232 driver. Therefore a suitable driver should be used with **Hrsout**. See **Hrsin** for circuits.

See also :**Declare**, **Rsin**, **Rsout**, **Serin**, **Serout**, **Hrsin**, **Hserin**, **Hserout**.

Hserin

Syntax

Hserin *Timeout*, *Timeout Label*, *Parity Error Label*, [*Modifiers*, *Variable* {, *Variable...* }]

Overview

Receive one or more values from the serial port on devices that contain a hardware USART. (Compatible with the melabs compiler)

Operators

Timeout is an *optional* value for the length of time the **Hserin** command will wait before jumping to label **Timeout Label**. **Timeout** is specified in 1 millisecond units and has a maximum of 16-bits.

Timeout Label is an optional valid BASIC label where **Hserin** will jump to in the event that a character has not been received within the time specified by **Timeout**.

Parity Error Label is an optional valid BASIC label where **Hserin** will jump to in the event that a Parity error is received. Parity is set using **Declares**. Parity Error detecting is not supported in the inline version of **Hserin** (first syntax example above).

Modifier is one of the many formatting modifiers, explained below.

Variable is a **Bit**, **Byte**, **Word**, or **Dword** variable, that will be loaded by **Hserin**.

Example

```
' Receive values serially and timeout if no reception after 1 second
Device 16F877
Declare Xtal = 4

Declare Hserial_Baud = 9600           ' Set baud rate to 9600
Declare Hserial_RCSTA = %10010000    ' Enable continuous receive
Declare Hserial_TXSTA = %00100000    ' Enable transmit and asynchronous mode
Declare Hserial_Clear = On           ' Clear the buffer before receiving
Dim Var1 as Byte

Loop:
  Hserin 1000, Timeout, [Var1] ' Receive a byte serially into Var1
  Print Dec Var1, " "          ' Display the byte received
  Goto Loop                    ' Loop forever

Timeout:
  Cls
  Print "Timed Out"           ' Display an error if Hserin timed out
  Stop
```

Hserin Modifiers.

As we already know, **Hserin** will wait for and receive a single byte of data, and store it in a variable. If the microcontroller was connected to a PC running a terminal program and the user pressed the "A" key on the keyboard, after the **Hserin** command executed, the variable would contain 65, which is the ASCII code for the letter "A"

What would happen if the user pressed the "1" key? The result would be that the variable would contain the value 49 (the ASCII code for the character "1"). This is an important point to remember: every time you press a character on the keyboard, the computer receives the ASCII value of that character. It is up to the receiving side to interpret the values as necessary. In this case, perhaps we actually wanted the variable to end up with the value 1, rather than the ASCII code 49.

The **Hserin** command provides a modifier, called the decimal modifier, which will interpret this for us. Look at the following code: -

```
Dim SerData as Byte
Hserin [Dec SerData]
```

Notice the decimal modifier in the **Hserin** command that appears just to the left of the SerData variable. This tells **Hserin** to convert incoming text representing decimal numbers into true decimal form and store the result in SerData. If the user running the terminal software pressed the "1", "2" and then "3" keys followed by a space or other non-numeric text, the value 123 will be stored in the variable SerData, allowing the rest of the program to perform any numeric operation on the variable.

Without the decimal modifier, however, you would have been forced to receive each character ("1", "2" and "3") separately, and then would still have to do some manual conversion to arrive at the number 123 (one hundred twenty three) before you can do the desired calculations on it.

The decimal modifier is designed to seek out text that represents decimal numbers. The characters that represent decimal numbers are the characters "0" through "9". Once the **Hserin** command is asked to use the decimal modifier for a particular variable, it monitors the incoming serial data, looking for the first decimal character. Once it finds the first decimal character, it will continue looking for more (accumulating the entire multi-digit number) until it finds a non-decimal numeric character. Remember that it will not finish until it finds at least one decimal character followed by at least one non-decimal character.

To illustrate this further, examine the following examples (assuming we're using the same code example as above): -

Serial input: "ABC"

Result: The program halts at the **Hserin** command, continuously waiting for decimal text.

Serial input: "123" (with no characters following it)

Result: The program halts at the **Hserin** command. It recognises the characters "1", "2" and "3" as the number one hundred twenty three, but since no characters follow the "3", it waits continuously, since there's no way to tell whether 123 is the entire number or not.

Serial input: "123" (followed by a space character)

Result: Similar to the above example, except once the space character is received, the program knows the entire number is 123, and stores this value in SerData. The **Hserin** command then ends, allowing the next line of code to run.

Serial input: "123A"

Result: Same as the example above. The "A" character, just like the space character, is the first non-decimal text after the number 123, indicating to the program that it has received the entire number.

Serial input: "ABCD123EFGH"

Result: Similar to examples 3 and 4 above. The characters "ABCD" are ignored (since they're not decimal text), the characters "123" are evaluated to be the number 123 and the following character, "E", indicates to the program that it has received the entire number.

The final result of the **Dec** modifier is limited to 16 bits (up to the value 65535). If a value larger than this is received by the decimal modifier, the end result will be incorrect because the

result rolled-over the maximum 16-bit value. Therefore, **Hserin** modifiers may not (at this time) be used to load **Dword** (32-bit) variables.

The decimal modifier is only one of a family of conversion modifiers available with **Hserin**. See below for a list of available conversion modifiers. All of the conversion modifiers work similar to the decimal modifier (as described above). The modifiers receive bytes of data, waiting for the first byte that falls within the range of characters they accept (e.g., "0" or "1" for binary, "0" to "9" for decimal, "0" to "9" and "A" to "F" for hex). Once they receive a numeric character, they keep accepting input until a non-numeric character arrives, or in the case of the fixed length modifiers, the maximum specified number of digits arrives.

While very effective at filtering and converting input text, the modifiers aren't completely fool-proof. As mentioned before, many conversion modifiers will keep accepting text until the first non-numeric text arrives, even if the resulting value exceeds the size of the variable. After **Hserin**, a **Byte** variable will contain the lowest 8 bits of the value entered and a **Word** (16-bits) would contain the lowest 16 bits. You can control this to some degree by using a modifier that specifies the number of digits, such as **Dec2**, which would accept values only in the range of 0 to 99.

Conversion Modifier	Type of Number	Numeric	Characters Accepted
Dec {1..10}	Decimal, optionally limited to 1 - 10 digits		0 through 9
Hex {1..8}	Hexadecimal, optionally limited to 1 - 8 digits		0 through 9, A through F
Bin {1..32}	Binary, optionally limited to 1 - 32 digits		0, 1

A variable preceded by **Bin** will receive the ASCII representation of its binary value. For example, if **Bin** Var1 is specified and "1000" is received, Var1 will be set to 8.

A variable preceded by **Dec** will receive the ASCII representation of its decimal value. For example, if **Dec** Var1 is specified and "123" is received, Var1 will be set to 123.

A variable preceded by **Hex** will receive the ASCII representation of its hexadecimal value. For example, if **Hex** Var1 is specified and "FE" is received, Var1 will be set to 254.

SKIP followed by a count will skip that many characters in the input stream. For example, **SKIP** 4 will skip 4 characters.

The **Hserin** command can be configured to wait for a specified sequence of characters before it retrieves any additional input. For example, suppose a device attached to the PICmicro™ is known to send many different sequences of data, but the only data you wish to observe happens to appear right after the unique characters, "XYZ". A modifier named **Wait** can be used for this purpose: -

```
Hserin [Wait("XYZ"), SerData]
```

The above code waits for the characters "X", "Y" and "Z" to be received, in that order, then it receives the next data byte and places it into variable SerData.

Str modifier.

The **Hserin** command also has a modifier for handling a string of characters, named **Str**.

The **Str** modifier is used for receiving a string of characters into a byte array variable.

A string is a set of characters that are arranged or accessed in a certain order. The characters "ABC" would be stored in a string with the "A" first, followed by the "B" then followed by the "C". A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string "ABC" would be stored in a byte array containing three bytes (elements).

Below is an example that receives ten bytes and stores them in the 10-byte array, SerString: -

```
Dim SerString[10] as Byte      ' Create a 10-byte array.
Hserin [Str SerString]        ' Fill the array with received data.
Print Str SerString           ' Display the string.
```

If the amount of received characters is not enough to fill the entire array, then a formatter may be placed after the array's name, which will only receive characters until the specified length is reached. For example: -

```
Dim SerString[10] as Byte      ' Create a 10-byte array.
Hserin [Str SerString\5]      ' Fill the first 5-bytes of the array
Print Str SerString\5         ' Display the 5-character string.
```

The example above illustrates how to fill only the first n bytes of an array, and then how to display only the first n bytes of the array. n refers to the value placed after the backslash.

Because of its complexity, serial communication can be rather difficult to work with at times. Using the guidelines below when developing a project using the **Hserin** and **Hserout** commands may help to eliminate some obvious errors: -

Always build your project in steps.

Start with small, manageable pieces of code, (that deal with serial communication) and test them, one individually.

Add more and more small pieces, testing them each time, as you go.

Never write a large portion of code that works with serial communication without testing its smallest workable pieces first.

Pay attention to timing.

Be careful to calculate and overestimate the amount of time, operations should take within the PICmicro™ for a given oscillator frequency. Misunderstanding the timing constraints is the source of most problems with code that communicate serially. If the serial communication in your project is bi-directional, the above statement is even more critical.

Pay attention to wiring.

Take extra time to study and verify serial communication wiring diagrams. A mistake in wiring can cause strange problems in communication, or no communication at all. Make sure to connect the ground pins (Vss) between the devices that are communicating serially.

Verify port setting on the PC and in the Hserin / Hserout commands.

Unmatched settings on the sender and receiver side will cause garbled data transfers or no data transfers. This is never more critical than when a line transceiver is used(i.e. MAX232). Always remember that a line transceiver inverts the serial polarity.

If the serial data received is unreadable, it is most likely caused by a baud rate setting error, or a polarity error.

If receiving data from another device that is not a PICmicro™, try to use baud rates of 9600 and below, or alternatively, use a higher frequency crystal.

Because of additional overheads in the PICmicro™, and the fact that the **Hserin** command offers a 2 level hardware receive buffer for serial communication, received data may sometimes be missed or garbled. If this occurs, try lowering the baud rate, or increasing the crystal frequency. Using simple variables (not arrays) will also increase the chance that the PICmicro™ will receive the data properly.

Declares

There are five Declare directives for use with **Hserin** . These are: -

Declare Hserial_Baud Constant value

Sets the BAUD rate that will be used to receive a value serially. The baud rate is calculated using the **Xtal** frequency declared in the program. The default baud rate if the Declare is not included in the program listing is 2400 baud.

Declare Hserial_RCSTA Constant value (0 to 255)

Hserial_RCSTA, sets the respective PICmicro™ hardware register RCSTA, to the value in the Declare. See the Microchip data sheet for the device used for more information regarding this register.

Declare Hserial_TXSTA Constant value (0 to 255)

Hserial_TXSTA, sets the respective PICmicro™ hardware register, TXSTA, to the value in the Declare. See the Microchip data sheet for the device used for more information regarding this register. The TXSTA register BRGH bit (bit 2) controls the high speed mode for the baud rate generator. Certain baud rates at certain oscillator speeds require this bit to be set to operate properly. To do this, set **Hserial_TXSTA** to a value of 24h instead of the normal 20h. Refer to the Microchip data sheet for the hardware serial port baud rate tables and additional information.

Declare Hserial_Parity Odd or Even

Enables/Disables parity on the serial port. For both **Hserin** and Hrsout The default serial data format is 8N1, 8 data bits, no parity bit and 1 stop bit. 7E1 (7 data bits, even parity, 1 stop bit) or 7O1 (7data bits, odd parity, 1 stop bit) may be enabled using the **Hserial_Parity** declare.

```
Declare Hserial_Parity = Even      ' Use if even parity desired
Declare Hserial_Parity = Odd       ' Use if odd parity desired
```

Declare Hserial_Clear On or Off

Clear the overflow error bit before commencing a read.

Because the hardware serial port only has a 2-byte input buffer, it can easily overflow is characters are not read from it often enough. When this occurs, the USART stops accepting any new characters, and requires resetting. This overflow error can be reset by strobing the CREN bit within the RCSTA register.

Example: -

```
RCSTA.4 = 0  
RCSTA.4 = 1
```

or

```
Clear RCSTA.4  
Set RCSTA.4
```

Alternatively, the **Hserial_Clear** declare can be used to automatically clear this error, even if no error occurred. However, the program will not know if an error occurred while reading, therefore some characters may be lost.

```
Declare Hserial_Clear = On
```

Notes

Hserin can only be used with devices that contain a hardware USART. See the specific device's data sheet for further information concerning the serial input pin as well as other relevant parameters.

Since the serial transmission is done in hardware, it is not possible to set the levels to an inverted state to eliminate an RS232 driver. Therefore a suitable driver should be used with **Hserin** . See **Hrsin** for suitable circuits.

See also : Declare, Hserout, Hrsin, Hrsout, Rsin, Rsout, Serin, Serout.

Hserout

Syntax

Hserout [*Item* {, *Item*... }]

Overview

Transmit one or more *Items* from the hardware serial port on devices that support asynchronous serial communications in hardware.

Operators

Item may be a constant, variable, expression, string list, or inline command.

There are no operators as such, instead there are *modifiers*. For example, if an at sign '@' precedes an *Item*, the ASCII representation for each digit is transmitted.

The modifiers are listed below: -

Modifier	Operation
----------	-----------

At ypos,xpos	Position the cursor on a serial LCD
Cls	Clear a serial LCD (also creates a 30ms delay)
Bin{1..32}	Send binary digits
Dec{1..10}	Send decimal digits
Hex{1..8}	Send hexadecimal digits
Sbin{1..32}	Send signed binary digits
Sdec{1..10}	Send signed decimal digits
Shex{1..8}	Send signed hexadecimal digits
Ibin{1..32}	Send binary digits with a preceding '%' identifier
Idec{1..10}	Send decimal digits with a preceding '#' identifier
Ihex{1..8}	Send hexadecimal digits with a preceding '\$' identifier
ISbin{1..32}	Send signed binary digits with a preceding '%' identifier
ISdec{1..10}	Send signed decimal digits with a preceding '#' identifier
IShex{1..8}	Send signed hexadecimal digits with a preceding '\$' identifier
Rep c\n	Send character c repeated n times
Str array\n	Send all or part of an array
Cstr cdata	Send string data defined in a Cdata statement.

The numbers after the **Bin**, **Dec**, and **Hex** modifiers are optional. If they are omitted, then the default is all the digits that make up the value will be displayed.

If a floating point variable is to be displayed, then the digits after the **Dec** modifier determine how many remainder digits are send. i.e. numbers after the decimal point.

```
Dim Flt as Float
Flt = 3.145
Hserout [Dec2 Flt]    ' Send 2 values after the decimal point
```

The above program will send 3.14

If the digit after the **Dec** modifier is omitted, then 3 values will be displayed after the decimal point.

```
Dim Flt as Float
Flt = 3.1456
Hserout [Dec Flt] ' Send 3 values after the decimal point
```

The above program will send 3.145

There is no need to use the **Sdec** modifier for signed floating point values, as the compiler's **Dec** modifier will automatically display a minus result: -

```
Dim Flt as Float
Flt = -3.1456
Hserout [Dec Flt] ' Send 3 values after the decimal point
```

The above program will send -3.145

Hex or **Bin** modifiers cannot be used with floating point values or variables.

The Xpos and Ypos values in the **At** modifier both start at 1. For example, to place the text "HELLO WORLD" on line 1, position 1, the code would be: -

```
Hserout [At 1, 1, "HELLO WORLD"]
```

Example 1

```
Dim Var1 as Byte
Dim Wrđ as Word
Dim Dwd as Dword

Hserout ["Hello World"] ' Display the text "Hello World"
Hserout ["Var1= ", Dec Var1] ' Display the decimal value of Var1
Hserout ["Var1= ", Hex Var1] ' Display the hexadecimal value of Var1
Hserout ["Var1= ", Bin Var1] ' Display the binary value of Var1
' Display 6 hex characters of a Dword type variable
Hserout ["Dwd= ", Hex6 Dwd]
```

Example 2

```
' Display a negative value on a serial LCD.
Symbol Negative = -200
Hserout [At 1, 1, Sdec Negative]
```

Example 3

```
' Display a negative value on a serial LCD with a preceding identifier.
Hserout [At 1, 1, IShex -$1234]
```

Example 3 will produce the text "\$-1234" on the LCD.

Some PICmicros such as the 16F87x, and 18FXXX range have the ability to read and write to their own flash memory. And although writing to this memory too many times is unhealthy for the PICmicro™, reading this memory is both fast, and harmless.

Which offers a unique form of data storage and retrieval, the Cdata command proves this, as it uses the mechanism of reading and storing in the PICmicro's flash memory.

Combining the unique features of the 'self modifying PICmicro's' with a string format, the compiler is capable of reducing the overhead of printing, or transmitting large amounts of text data. The **Cstr** modifier may be used in commands that deal with text processing i.e. Serout, Hrsout, and Print etc.

The **Cstr** modifier is used in conjunction with the Cdata command. The Cdata command is used for initially creating the string of characters: -

```
String1: Cdata "HELLO WORLD", 0
```

The above line of code will create, in flash memory, the values that make up the ASCII text "HELLO WORLD", at address String1. Note the null terminator after the ASCII text.

null terminated means that a zero (null) is placed at the end of the string of ASCII characters to signal that the string has finished.

To display, or transmit this string of characters, the following command structure could be used:

```
Hserout [Cstr String1]
```

The label that declared the address where the list of Cdata values resided, now becomes the string's name. In a large program with lots of text formatting, this type of structure can save quite literally hundreds of bytes of valuable code space.

Try both these small programs, and you'll see that using **Cstr** saves a few bytes of code: -

First the standard way of displaying text: -

```
Device = 16F877
Cls
Hserout ["HELLO WORLD",13]
Hserout ["HOW ARE YOU?",13]
Hserout ["I AM FINE!",13]
Stop
```

Now using the **Cstr** modifier: -

```
Cls
Hserout [Cstr TEXT1]
Hserout [Cstr TEXT2]
Hserout [Cstr TEXT3]
Stop
```

```
TEXT1: Cdata "HELLO WORLD", 13, 0
TEXT2: Cdata "HOW ARE YOU?", 13, 0
TEXT3: Cdata "I AM FINE!", 13, 0
```

Again, note the null terminators after the ASCII text in the Cdata commands. Without these, the PICmicro™ will continue to transmit data in an endless loop.

The term 'virtual string' relates to the fact that a string formed from the **Cdata** command cannot be written too, but only read from.

The **Str** modifier is used for sending a string of bytes from a byte array variable. A string is a set of bytes sized values that are arranged or accessed in a certain order. The values 1, 2, 3 would be stored in a string with the value 1 first, followed by 2 then followed by the value 3. A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string 1,2,3 would be stored in a byte array containing three bytes (elements).

Below is an example that displays four bytes (from a byte array): -

```
Dim MyArray[10] as Byte      ' Create a 10-byte array.
MyArray [0] = "H"           ' Load the first 5 bytes of the array
MyArray [1] = "E"           ' With the data to send
MyArray [2] = "L"
MyArray [3] = "L"
MyArray [4] = "O"
Hserout [Str MyArray\5]     ' Display a 5-byte string.
```

Note that we use the optional `\n` argument of **Str**. If we didn't specify this, the PICmicro™ would try to keep sending characters until all 10 bytes of the array were transmitted. Since we do not wish all 10 bytes to be transmitted, we chose to tell it explicitly to only send the first 5 bytes.

The above example may also be written as: -

```
Dim MyArray [10] as Byte    ' Create a 10-byte array.
Str MyArray = "HELLO"       ' Load the first 5 bytes of the array
Hserout [Str MyArray\5]     ' Send 5-byte string.
```

The above example, has exactly the same function as the previous one. The only difference is that the string is now constructed using **Str** as a command instead of a modifier.

Declares

There are four Declare directives for use with **Hserout**. These are: -

Declare Hserial_Baud Constant value

Sets the BAUD rate that will be used to transmit a value serially. The baud rate is calculated using the **Xtal** frequency declared in the program. The default baud rate if the **Declare** is not included in the program listing is 2400 baud.

Declare Hserial_RCSTA Constant value (0 to 255)

Hserial_RCSTA, sets the respective PICmicro™ hardware register RCSTA, to the value in the **Declare**. See the Microchip data sheet for the device used for more information regarding this register. Refer to the upgrade manual pages for a description of the RCSTA register.

Declare Hserial_TXSTA Constant value (0 to 255)

Hserial_TXSTA, sets the respective PICmicro™ hardware register, TXSTA, to the value in the **Declare**. See the Microchip data sheet for the device used for more information regarding this register. The TXSTA register BRGH bit (bit 2) controls the high speed mode for the baud rate generator. Certain baud rates at certain oscillator speeds require this bit to be set to operate properly. To do this, set **Hserial_TXSTA** to a value of 24h instead of the normal 20h.

Refer to the Microchip data sheet for the hardware serial port baud rate tables and additional information. Refer to the upgrade manual pages for a description of the TXSTA register.

Declare Hserial_Parity Odd or Even

Enables/Disables parity on the serial port. For both **Hserout** and **Hserin** The default serial data format is 8N1, 8 data bits, no parity bit and 1 stop bit. 7E1 (7 data bits, even parity, 1 stop bit) or 7O1 (7data bits, odd parity, 1 stop bit) may be enabled using the **Hserial_Parity** declare.

```
Declare Hserial_Parity = Even    ' Use if even parity desired
Declare Hserial_Parity = Odd     ' Use if odd parity desired
```

Notes

Hserout can only be used with devices that contain a hardware USART. See the specific device's data sheet for further information concerning the serial input pin as well as other relevant parameters.

Since the serial transmission is done in hardware, it is not possible to set the levels to an inverted state in order to eliminate an RS232 driver. Therefore a suitable driver should be used with **Hserout** . See **Hrsin** for circuit examples

See also : **Declare**, **Rsin**, **Rsout**, **Serin**, **Serout**, **Hserin**, **Hserin**.

I2Cin

Syntax

I2Cin *Dpin*, *Cpin*, *Control*, { *Address* }, [*Variable* {, *Variable*...}]

Overview

Receives a value from the I²C bus, and places it into *variable/s*.

Operators

Dpin is a Port.Pin constant that specifies the I/O pin that will be connected to the I²C device's data line (SDA). This pin's I/O direction will be changed to input and will remain in that state after the instruction is completed.

Cpin is a Port.Pin constant that specifies the I/O pin that will be connected to the I²C device's clock line (SCL). This pin's I/O direction will be changed to output.

Variable is a user defined variable of type **Bit**, **Byte**, **Word**, **Dword**, **Float**, **Array**.

Control is a constant value or a byte sized variable expression.

Address is an optional constant value or a variable expression.

The **I2Cin** command operates as an I²C master, and may be used to interface with any device that complies with the 2-wire I²C protocol. The most significant 7-bits of control byte contain the control code and the slave address of the device being interfaced with. Bit-0 is the flag that indicates whether a read or write command is being implemented.

For example, if we were interfacing to an external eeprom such as the 24LC32, the control code would be %10100001 or \$A1. The most significant 4-bits (1010) are the eeprom's unique slave address. Bits 1 to 3 reflect the three address pins of the eeprom. And bit-0 is set to signify that we wish to read from the eeprom. Note that this bit is automatically set by the **I2Cin** command, regardless of its initial setting.

Example

```
' Receive a byte from the I2C bus and place it into variable Var1.
Dim Var1 as Byte           ' We'll only read 8-bits
Dim Address as Word       ' 16-bit address required
Symbol Control %10100001  ' Target an eeprom
Symbol SDA = PORTC.3      ' Alias the SDA (Data) line
Symbol SCL = PORTC.4      ' Alias the SCL (Clock) line
Address = 20              ' Read the value at address 20
I2Cin SDA, SCL, Control, Address, [Var1] ' Read the byte from the eeprom
```

Address is an optional parameter that may be an 8-bit or 16-bit value. If a variable is used in this position, the size of address is dictated by the size of the variable used (byte or word). In the case of the previous eeprom interfacing, the 24LC32 eeprom requires a 16-bit address. While the smaller types require an 8-bit address. Make sure you assign the right size address for the device interfaced with, or you may not achieve the results you intended.

The **I2Cin** command allows differing variable assignments. For example: -

```
Dim Var1 as Byte
Dim Wrd as Word
I2Cin SDA, SCL, Control, Address, [Var1, Wrd]
```

The above example will receive two values from the bus, the first being an 8-bit value dictated by the size of variable Var1 which has been declared as a byte. And a 16-bit value, this time dictated by the size of the variable Wrd which has been declared as a word. Of course, bit type variables may also be used, but in most cases these are not of any practical use as they still take up a byte within the eeprom.

Declares

See **I2Cout** for declare explanations.

Notes

When the **I2Cin** command is used, the appropriate SDA and SCL Port and Pin are automatically setup as inputs, and outputs. Because the I²C protocol calls for an open-collector interface, pull-up resistors are required on both the SDA and SCL lines. Values of 4.7KΩ to 10KΩ will suffice.

Str modifier with I2Cin

Using the **Str** modifier allows the **I2Cin** command to transfer the bytes received from the I²C bus directly into a byte array. If the amount of received characters is not enough to fill the entire array, then a formatter may be placed after the array's name, which will only receive characters until the specified length is reached. An example of each is shown below: -

```
Dim Array[10] as Byte           ' Define an array of 10 bytes
Dim Address as Byte            ' Create a word sized variable
' Load data into all the array
I2Cin SDA, SCL, %10100000, Address, [Str Array]
' Load data into only the first 5 elements of the array
I2Cin SDA, SCL, %10100000, Address, [Str Array\5]
```

See Also: **BusAck, Bstart, Brestart, Bstop, Busout, HbStart, HbRestart, HbusAck, Hbusin, Hbusout, I2Cout**

I2Cout

Syntax

I2Cout *Control*, { *Address* }, [*OutputData*]

Overview

Transmit a value to the I²C bus, by first sending the *control* and optional *address* out of the clock pin (*SCL*), and data pin (*SDA*).

Operators

Dpin is a Port.Pin constant that specifies the I/O pin that will be connected to the I²C device's data line (*SDA*). This pin's I/O direction will be changed to input and will remain in that state after the instruction is completed.

Cpin is a Port.Pin constant that specifies the I/O pin that will be connected to the I²C device's clock line (*SCL*). This pin's I/O direction will be changed to output.

Control is a constant value or a byte sized variable expression.

Address is an optional constant, variable, or expression.

OutputData is a list of variables, constants, expressions and modifiers that informs I2Cout how to format outgoing data. **I2Cout** can transmit individual or repeating bytes, convert values into decimal, hex or binary text representations, or transmit strings of bytes from variable arrays.

These actions can be combined in any order in the *OutputData* list.

The **I2Cout** command operates as an I²C master and may be used to interface with any device that complies with the 2-wire I²C protocol. The most significant 7-bits of *control* byte contain the control code and the slave address of the device being interfaced with. Bit-0 is the flag that indicates whether a read or write command is being implemented.

For example, if we were interfacing to an external eeprom such as the 24LC32, the control code would be %10100000 or \$A0. The most significant 4-bits (1010) are the eeprom's unique slave address. Bits 1 to 3 reflect the three address pins of the eeprom. And Bit-0 is clear to signify that we wish to write to the eeprom. Note that this bit is automatically cleared by the **I2Cout** command, regardless of its initial value.

Example

```
' Send a byte to the I2C bus.
Dim Var1 as Byte           ' We'll only read 8-bits
Dim Address as Word       ' 16-bit address required
Symbol Control = %10100000 ' Target an eeprom
Symbol SDA = PORTC.3      ' Alias the SDA (Data) line
Symbol SCL = PORTC.4      ' Alias the SCL (Clock) line
Address = 20              ' Write to address 20
Var1 = 200               ' The value place into address 20
I2Cout SDA, SCL, Control, Address, [Var1] ' Send the byte to the eeprom
DelayMs 10                ' Allow time for allocation of byte
```

Address is an optional parameter that may be an 8-bit or 16-bit value. If a variable is used in this position, the size of *address* is dictated by the size of the variable used (byte or word). In the case of the above eeprom interfacing, the 24LC32 eeprom requires a 16-bit address. While the smaller types require an 8-bit address. Make sure you assign the right size address for the device interfaced with, or you may not achieve the results you intended.

The value sent to the bus depends on the size of the variables used. For example: -

```
Dim Wrd as Word           ' Declare a Word size variable
I2Cout SDA, SCL, Control, Address, [Wrd]
```

Will send a 16-bit value to the bus. While: -

```
Dim Var1 as Byte         ' Declare a Byte size variable
I2Cout SDA, SCL, Control, Address, [Var1]
```

Will send an 8-bit value to the bus. Using more than one variable within the brackets allows differing variable sizes to be sent. For example: -

```
Dim Var1 as Byte
Dim Wrd as Word
I2Cout SDA, SCL, Control, Address, [Var1, Wrd]
```

Will send two values to the bus, the first being an 8-bit value dictated by the size of variable Var1 which has been declared as a byte. And a 16-bit value, this time dictated by the size of the variable Wrd which has been declared as a word. Of course, bit type variables may also be used, but in most cases these are not of any practical use as they still take up a byte within the eeprom.

A string of characters can also be transmitted, by enclosing them in quotes: -

```
I2Cout SDA, SCL, Control, Address, ["Hello World", Var1, Wrd]
```

Str modifier with I2Cout

The **Str** modifier is used for transmitting a string of bytes from a byte array variable. A string is a set of bytes sized values that are arranged or accessed in a certain order. The values 1, 2, 3 would be stored in a string with the value 1 first, followed by 2 then followed by the value 3. A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string 1,2,3 would be stored in a byte array containing three bytes (elements). Below is an example that sends four bytes from an array: -

```
Dim MyArray[10] as Byte           ' Create a 10-byte array.
MyArray [0] = "A"                 ' Load the first 4 bytes of the array
MyArray [1] = "B"                 ' With the data to send
MyArray [2] = "C"
MyArray [3] = "D"
' Send a 4-byte string
I2Cout SDA, SCL, %10100000, Address, [Str MyArray\4]
```

Note that we use the optional \n argument of **Str**. If we didn't specify this, the program would try to keep sending characters until all 10 bytes of the array were transmitted. Since we do not wish all 10 bytes to be transmitted, we chose to tell it explicitly to only send the first 4 bytes.

Declares

There are two **Declare** directives for use with **I2Cout**. These are: -

Declare I2C_Slow_Bus On - Off or 1 – 0

Slows the bus speed when using an oscillator higher than 4MHz. The standard speed for the I²C bus is 100KHz. Some devices use a higher bus speed of 400KHz. If you use an 8MHz or higher oscillator, the bus speed may exceed the devices specs, which will result in intermittent transactions, or in some cases, no transactions at all. Therefore, use this **Declare** if you are not sure of the device's spec. The datasheet for the device used will inform you of its bus speed.

Declare I2C_Bus_SCL On - Off, 1 - 0 or **True - False**

Eliminates the necessity for a pullup resistor on the SCL line.

The I²C protocol dictates that a pullup resistor is required on both the SCL and SDA lines, however, this is not always possible due to circuit restrictions etc, so once the **I2C_Bus_SCL On Declare** is issued at the top of the program, the resistor on the SCL line can be omitted from the circuit. The default for the compiler if the **I2C_Bus_SCL Declare** is not issued, is that a pullup resistor is required.

Notes

When the **I2Cout** command is used, the appropriate SDA and SCL Port and Pin are automatically setup as inputs, and outputs. Because the I²C protocol calls for an *open-collector* interface, pull-up resistors are required on both the SDA and SCL lines. Values of 4.7K Ω to 10K Ω will suffice.

You may imagine that it's limiting having a fixed set of pins for the I²C interface, but you must remember that several different devices may be attached to a single bus, each having a unique slave address. Which means there is usually no need to use up more than two pins on the PICmicro™ in order to interface to many devices.

See Also: **BusAck, Bstart, Brestart, Bstop, Busin, HbStart, HbRestart, HbusAck, Hbusin, Hbusout, I2Cin**

If..Then..Elseif..Else..EndIf

Syntax

If *Comparison* **Then** *Instruction* : { *Instruction* }

Or, you can use the single line form syntax:

If *Comparison* **Then** *Instruction* : { *Instruction* } : **Elseif** *Comparison* **Then** *Instruction* : **Else** *Instruction*

Or, you can use the block form syntax:

```
If Comparison Then  
Instruction(s)  
Elseif Comparison Then  
Instruction(s)  
{  
Elseif Comparison Then  
Instruction(s)  
}  
Else  
Instruction(s)  
EndIf
```

The curly braces signify optional conditions.

Overview

Evaluates the *comparison* and, if it fulfils the criteria, executes *expression*. If *comparison* is not fulfilled the *instruction* is ignored, unless an **Else** directive is used, in which case the code after it is implemented until the **EndIf** is found.

When all the instruction are on the same line as the **If-Then** statement, all the instructions on the line are carried out if the condition is fulfilled.

Operators

Comparison is composed of variables, numbers and comparators.

Instruction is the statement to be executed should the *comparison* fulfil the **If** criteria

Example 1

```
Symbol LED = PORTB.4  
Var1 = 3  
Low LED  
If Var1 > 4 Then High LED : DelayMs 500 : Low LED
```

In the above example, Var1 is not greater than 4 so the **If** criteria isn't fulfilled. Consequently, the **High** LED statement is never executed leaving the state of port pin PORTB.4 low. However, if we change the value of variable Var1 to 5, then the LED will turn on for 500ms then off, because Var1 is now greater than 4, so fulfils the *comparison* criteria.

A second form of **If**, evaluates the expression and if it is true then the first block of instructions is executed. If it is false then the second block (after the **Else**) is executed.

The program continues after the **EndIf** instruction.

The **Else** is optional. If it is missed out then if the expression is false the program continues after the **EndIf** line.

Example 2

```
If X & 1 = 0 Then
    A = 0
    B = 1
Else
    A = 1
EndIf
If Z = 1 Then
    A = 0
    B = 0
EndIf
```

Example 3

```
If X = 10 Then
    High LED1
ElseIf X = 20 Then
    High LED2
Else
    High LED3
EndIf
```

A forth form of **If**, allows the **Else** or **Elseif** to be placed on the same line as the **If**: -

```
If X = 10 Then High LED1 : ElseIf X = 20 Then High LED2 : Else : High LED3
```

Notice that there is no **EndIf** instruction. The comparison is automatically terminated by the end of line condition. So in the above example, if X is equal to 10 then LED1 will illuminate, if X equals 20 then LED will illuminate, otherwise, LED3 will illuminate.

The **If** statement allows any type of variable, register or constant to be compared. A common use for this is checking a Port bit: -

```
If PORTA.0 = 1 Then High LED : Else : Low LED
```

Any commands on the same line after **Then** will only be executed if the comparison is fulfilled: -

```
If Var1 = 1 Then High LED : DelayMs 500 : Low LED
```

Notes

A **Goto** command is optional after the **Then**: -

```
If PORTB.0 = 1 Then Label
```

Then operand always required.

The Proton compiler relies heavily on the **Then** part. Therefore, if the **Then** part of a construct is left out of the code listing, a Syntax Error will be produced.

See also : **Boolean Logic Operators, Select..Case..EndSelect.**

Include

Syntax

Include "Filename"

Overview

Include another file at the current point in the compilation. All the lines in the new file are compiled as if they were in the current file at the point of the **Include** directive.

A common use for the include command is shown in the example below. Here a small master program is used to include a number of smaller library files which are all compiled together to make the overall program.

Operators

Filename is any valid Proton file.

Example

```
' Main Program Includes sub files
  Include "StartCode.bas"
  Include "MainCode.bas"
  Include "EndCode.bas"
```

Notes

The file to be included into the BASIC listing may be in one of three places on the hard drive if a specific path is not chosen.

- 1... Within the BASIC program's directory.
- 2... Within the Compiler's current directory.
- 3... Within the Includes\Sources folder of the compiler's current directory.

The list above also shows the order in which they are searched for.

Using Include files to tidy up your code.

If the include file contains assembler subroutines then it must always be placed at the beginning of the program. This allows the subroutine/s to be placed within the first bank of memory (0..2048), thus avoiding any bank boundary errors. Placing the include file at the beginning of the program also allows all of the variables used by the routines held within it to be pre-declared. This again makes for a tidier program, as a long list of variables is not present in the main program.

There are some considerations that must be taken into account when writing code for an include file, these are: -

- 1). Always jump over the subroutines.

When the include file is placed at the top of the program this is the first place that the compiler starts, therefore, it will run the subroutine/s first and the **Return** command will be pointing to a random place within the code. To overcome this, place a **Goto** statement just before the subroutine starts.

For example: -

```
Goto Over_This_Subroutine ' Jump over the subroutine  
' The subroutine is placed here
```

```
Over_This_Subroutine:      ' Jump to here first
```

2). Variable and Label names should be as meaningful as possible.

For example. Instead of naming a variable **Loop**, change it to **ISUB_Loop**. This will help eliminate any possible duplication errors, caused by the main program trying to use the same variable or label name. However, try not to make them too obscure as your code will be harder to read and understand, it might make sense at the time of writing, but come back to it after a few weeks and it will be meaningless.

3). Comment, Comment, and Comment some more.

This cannot be emphasised enough. Always place a plethora of remarks and comments. The purpose of the subroutine/s within the include file should be clearly explained at the top of the program, also, add comments after virtually every command line, and clearly explain the purpose of all variables and constants used. This will allow the subroutine to be used many weeks or months after its conception. A rule of thumb that I use is that I can understand what is going on within the code by reading only the comments to the right of the command lines.

Inc

Syntax

Inc *Variable*

Overview

Increment a variable i.e. $\text{Var1} = \text{Var1} + 1$

Operators

Variable is a user defined variable

Example

```
Var1 = 1
Repeat
  Print Dec Var1, " "
  DelayMs 200
  Inc Var1
Until Var1 > 10
```

The above example shows the equivalent to the For-Next loop: -

```
For Var1 = 1 to 10 : Next
```

See also :Dec.

Inkey

Syntax

Variable = **Inkey**

Overview

Scan a keypad and place the returned value into *variable*

Operators

Variable is a user defined variable

Example

```
Dim Var1 as Byte
Var1 = Inkey           ' Scan the keypad
DelayMs 50           ' Debounce by waiting 50ms
Print Dec Var1, " "  ' Display the result on the LCD
```

Notes

Inkey will return a value between 0 and 16. If no key is pressed, the value returned is 16.

Using a **LookUp** command, the returned values can be re-arranged to correspond with the legends printed on the keypad: -

```
Var1 = Inkey
KEY = LookUp Var1, [255,1,4,7,"*",2,5,8,0,3,6,9,"#",0,0,0]
```

The above example is only a demonstration, the values inside the **LookUp** command will need to be re-arranged for the type of keypad used, and its connection configuration.

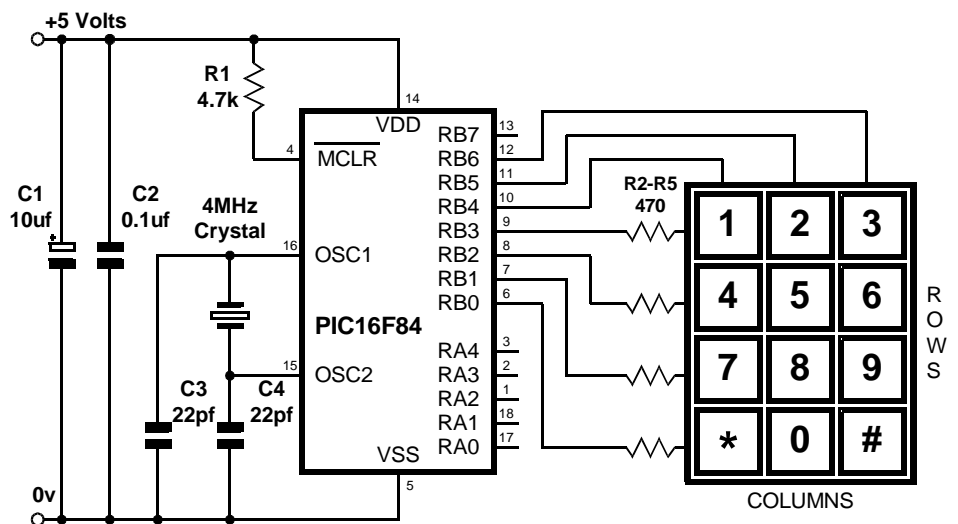
Declare

Declare Keypad_Port Port

Assigns the Port that the keypad is attached to.

The keypad routine requires pull-up resistors, therefore, the best Port for this device is PORTB, which comes equipped with internal pull-ups. If the Declare is not used in the program, then PORTB is the default Port.

The 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).



Input

Syntax

Input *Port* . *Pin*

Overview

Makes the specified *Port* or *Pin* an input.

Operators

Port.Pin must be a Port, or Port.Pin constant declaration.

Example

```
Input PORTA.0      ' Make bit-0 of PORTA an input
```

```
Input PORTA        ' Make all of PORTA an input
```

Notes

An Alternative method for making a particular pin an input is by directly modifying the Tris register: -

```
TRISB.0 = 1        ' Set PORTB, bit-0 to an input
```

All of the pins on a port may be set to inputs by setting the whole Tris register at once: -

```
TRISB = %11111111 ' Set all of PORTB to inputs
```

In the above examples, setting a Tris bit to 1 makes the pin an input, and conversely, setting the bit to 0 makes the pin an output.

See also :Output.

LCDread

Syntax

Variable = **LCDread** *Ypos*, *Xpos*

Or

Variable = **LCDread Text** *Ypos*, *Xpos*

Overview

Read a byte from a graphic LCD. Can also read Text RAM from a Toshiba T6963 LCD.

Operators

Variable is a user defined variable.

Ypos :-

With a Samsung KS0108 graphic LCD this may be a constant, variable or expression within the range of 0 to 7 This corresponds to the line number of the LCD, with 0 being the top row.

With a Toshiba T6963 graphic LCD this may be a constant, variable or expression within the range of 0 to the Y resolution of the display. With 0 being the top line.

Xpos :-

With a Samsung KS0108 graphic LCD this may be a constant, variable or expression with a value of 0 to 127. This corresponds to the X position of the LCD, with 0 being the far left column.

With a Toshiba graphic LCD this may be a constant, variable or expression with a value of 0 to the X resolution of the display divided by the font width ($LCD_X_Res / LCD_Font_Width$). This corresponds to the X position of the LCD, with 0 being the far left column.

Example

```
' Read and display the top row of the Samsung KS0108 graphic LCD
Device = 16F877
Declare LCD_Type = Graphic ' Target a Samsung graphic LCD

Dim Var1 as Byte
Dim Xpos as Byte
Cls ' Clear the LCD
Print "Testing 1 2 3"
For Xpos = 0 to 127 ' Create a loop of 128
    Var1 = LCDread 0, Xpos ' Read the LCD's top line
    Print At 1, 0, "Chr= ", Dec Var1, " "
    DelayMs 100
Next
Stop
```

Notes

The graphic LCDs that are compatible with Proton are the Samsung KS0108, and the Toshiba T6963. The Samsung display has a pixel resolution of 64 x 128. The 64 being the Y axis, made up of 8 lines each having 8-bits. The 128 being the X axis, made up of 128 positions. The Toshiba LCDs are available with differing resolutions.

As with **LCDwrite**, the graphic LCD must be targeted using the **LCD_Type Declare** directive before this command may be used.

The Toshiba T6963 graphic LCDs split their graphic and text information within internal RAM. This means that the **LCDread** command can also be used to read the textual information as well as the graphical information present on the LCD. Placing the word **Text** after the **LCDread** command will direct the reading process to Text RAM.

Example

```
' Read text from a Toshiba graphic LCD
Device = 18F452
Declare LCD_Type = Toshiba      ' Use a Toshiba T6963 graphic LCD
,
' LCD interface pin assignments
,
Declare LCD_DTPort = PORTD      ' LCD's Data port
Declare LCD_WRPin = PORTE.2    ' LCD's WR line
Declare LCD_RDPin = PORTE.1    ' LCD's RD line
Declare LCD_CEPin = PORTE.0    ' LCD's CE line
Declare LCD_CDPin = PORTA.1    ' LCD's CD line
Declare LCD_RSTPin = PORTA.0   ' LCD's RESet line (Optional)
,
' LCD characteristics
,
Declare LCD_X_Res = 128        ' LCD's X Resolution
Declare LCD_Y_Res = 64        ' LCD's Y Resolution
Declare LCD_Font_Width = 8    ' The width of the LCD's font

Dim Charpos as Byte           ' The X position of the read
Dim Char as Byte              ' The byte read from the LCD

DelayMs 100                   ' Wait for the LCD to stabilise
ADCON1 = 7                     ' PORTA and PORTE to all digital mode
Cls                             ' Clear the LCD
Print At 0,0," This is for Copying" ' Display text on top line of LCD
For Charpos = 0 to 20          ' Create a loop of 21 cycles
    Char = LCDread Text 0,Charpos ' Read the top line of the LCD
    Print At 1,Charpos,Char      ' Print the byte read on the second line
    DelayMs 100                  ' A small delay so we can see things happen
Next                             ' Close the loop
```

See also : **LCDwrite** for a description of the screen formats, **Pixel**, **Plot**, **Toshiba_Command**, **Toshiba_UDG**, **UnPlot**, see **Print** for LCD connections.

LCDwrite

Syntax

LCDwrite *Ypos*, *Xpos*, [*Value* ,{ *Value etc...* }]

Overview

Write a byte to a graphic LCD.

Operators

Ypos :-

With a **Samsung** KS0108 graphic LCD this may be a constant, variable or expression within the range of 0 to 7 This corresponds to the line number of the LCD, with 0 being the top row.

With a **Toshiba** T6963 graphic LCD this may be a constant, variable or expression within the range of 0 to the Y resolution of the display. With 0 being the top line.

Xpos :-

With a **Samsung** KS0108 graphic LCD this may be a constant, variable or expression with a value of 0 to 127. This corresponds to the X position of the LCD, with 0 being the far left column.

With a **Toshiba** graphic LCD this may be a constant, variable or expression with a value of 0 to the X resolution of the display divided by the font width ($LCD_X_Res / LCD_Font_Width$). This corresponds to the X position of the LCD, with 0 being the far left column.

Value may be a constant, variable, or expression, within the range of 0 to 255 (byte).

Example 1

```
' Display a line on the top row of a Samsung KS0108 graphic LCD
Device = 16F877
Declare LCD_Type = Graphic           ' Target a Samsung graphic LCD
Dim Xpos as Byte
Cls                                   ' Clear the LCD
For Xpos = 0 to 127                  ' Create a loop of 128
    LCDwrite 0, Xpos, [%11111111]    ' Write to the LCD's top line
    DelayMs 100
Next
Stop
```

Example 2

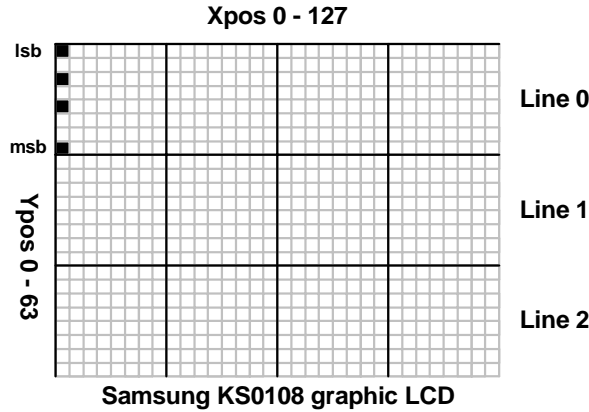
```
' Display a line on the top row of a Toshiba 128x64 graphic LCD
Device = 16F877
Declare LCD_Type = Toshiba           ' Target a Toshiba graphic LCD
Dim Xpos as Byte
Cls                                   ' Clear the LCD
For Xpos = 0 to 20                    ' Create a loop of 21
    LCDwrite 0, Xpos, [%00111111]    ' Write to the LCD's top line
    DelayMs 100
Next
Stop
```

Notes

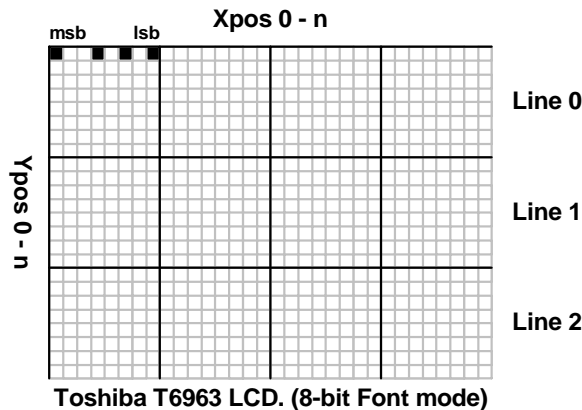
The graphic LCDs that are compatible with Proton are the Samsung KS0108, and the Toshiba T6963. The Samsung display has a pixel resolution of 64 x 128. The 64 being the Y axis, made up of 8 lines each having 8-bits. The 128 being the X axis, made up of 128 positions. The Toshiba LCDs are available with differing resolutions.

There are important differences between the Samsung and Toshiba screen formats. The diagrams below show these in more detail: -

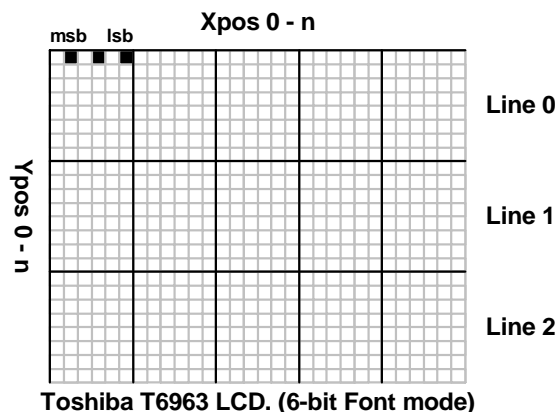
The diagram below illustrates the position of one byte at position 0,0 on a Samsung KS0108 LCD screen. The least significant bit is located at the top. The byte displayed has a value of 149 (10010101).



The diagram below illustrates the position of one byte at position 0,0 on a Toshiba T6963 LCD screen in 8-bit font mode. The least significant bit is located at the right of the screen byte. The byte displayed has a value of 149 (10010101).



The diagram below illustrates the position of one byte at position 0,0 on a Toshiba T6963 LCD screen in 6-bit font mode. The least significant bit is located at the right of the screen byte. The byte displayed still has a value of 149 (10010101), however, only the first 6 bits are displayed (010101) and the other two are discarded.



See also : [LCDread](#), [Plot](#), [Toshiba_Command](#), [Toshiba_UDG](#), [UnPlot](#)
see [Print for LCD connections](#).

Ldata

Syntax

Ldata { *alphanumeric data* }

Overview

Place information into code memory using the `Retlw` instruction when used with a standard 14-bit core devices, and Flash (code) memory when using an 18F or enhanced 14-bit core device. For access by **Lread**, **Lread8**, **Lread16** or **Lread32**.

Operators

alphanumeric data can be a 8,16, 32 bit value, or floating point values, or any alphabetic character or string enclosed in quotes.

Example

```
Device = 16F877
Dim Char as Byte
Dim Loop as Byte
Cls
For Loop = 0 to 9           ' Create a loop of 10
    Char = Lread Label + Loop ' Read memory location Label + Loop
    Print Char              ' Display the value read
Next
Stop
Label: Ldata "HELLO WORLD" ' Create a string of text in code memory
```

The program above reads and displays 10 values from the address located by the `Label` accompanying the **Ldata** command. Resulting in "HELLO WORL" being displayed.

Ldata is not simply used for character storage, it may also hold 8, 16, 32 bit, or floating point values. The example below illustrates this: -

```
Device = 16F628
Dim Var1 as Byte
Dim Wrld as Word
Dim Dwd1 as Dword
Dim Flt1 as Float
Cls
Var1 = Lread Bit8_Val      ' Read the 8-bit value
Print Dec Var1, " "
Wrld = Lread Bit16_Val     ' Read the 16-bit value
Print Dec Wrld
Dwd1 = Lread Bit32_Val     ' Read the 32-bit value
Print At 2,1, Dec Dwd1, " "
Flt1 = Lread Flt_Val      ' Read the floating point value
Print Dec Flt1
Stop
Bit8_Val: Ldata 123
Bit16_Val: Ldata 1234
Bit32_Val: Ldata 123456
Flt_Val: Ldata 123.456
```

Floating point examples.

14-bit core example

```
' 14-bit read floating point data from a table and display the results
Device = 16F877
Dim Flt as Float          ' Declare a Floating Point variable
Dim Fcount as Byte
Cls                        ' Clear the LCD
Fcount = 0                 ' Clear the table counter
Repeat                     ' Create a loop
  Flt = Lread FlTable + Fcount ' Read the data from the Ldata table
  Print At 1, 1, Dec3 Flt ' Display the data read
  Fcount = Fcount + 4      ' Point to next value, by adding 4 to counter
  DelayMs 1000             ' Slow things down
Until Flt = 0.005         ' Stop when 0.005 is read
Stop
FlTable:
Ldata as Float 3.14, 65535.123, 1234.5678, -1243.456, -3.14, 998999.12, _
               0.005
```

18F device example

```
' 18F read floating point data from a table and display the results
Device = 18F452
Dim Flt as Float          ' Declare a Floating Point variable
Dim Fcount as Byte
Cls                        ' Clear the LCD
Fcount = 0                 ' Clear the table counter
Repeat                     ' Create a loop
  Flt = Lread FlTable + Fcount ' Read the data from the Ldata table
  Print At 1, 1, Dec3 Flt ' Display the data read
  Fcount = Fcount + 2      ' Point to next value, by adding 2 to counter
  DelayMs 1000             ' Slow things down
Until Flt = 0.005         ' Stop when 0.005 is read
Stop
FlTable:
Ldata as Float 3.14, 65535.123, 1234.5678, -1243.456, -3.14, 998999.12, _
               0.005
```

Notes

Ldata tables should be placed at the end of the BASIC program. If an **Ldata** table is placed at the beginning of the program, then a **Goto** command must jump over the tables, to the main body of code.

```
Goto OverDataTable
```

```
Ldata 1,2,3,4,5,6
```

```
OverDataTable:
```

```
{ rest of code here }
```

With **14-bit** core devices, an 8-bit value (0 - 255) in an **Ldata** statement will occupy a single code space, however, 16-bit data (0 - 65535) will occupy two spaces, 32-bit and floating point values will occupy 4 spaces. This must be taken into account when using the **Lread** command. See 14-bit floating point example above.

With 18F devices, an 8, and 16-bit value in an **Ldata** statement will occupy a single code space, however, 32-bit and floating point values will occupy 2 spaces. This must be taken into account when using the **Lread** command. See 16-bit floating point example above.

18F device requirements.

The compiler uses a different method of holding information in an **Ldata** statement when using 18F devices. It uses the unique capability of these devices to read from their own code space, which offers optimisations when values larger than 8-bits are stored. However, because the 18F devices are **Byte** oriented, as opposed to the 14-bit types which are **Word** oriented. The **Ldata** tables should contain an even number of values, or corruption may occur on the last value read. For example: -

```
Even: Ldata 1,2,3,"123"
```

```
Odd: Ldata 1,2,3,"12"
```

An **Ldata** table containing an Odd amount of values will produce a compiler WARNING message.

Formatting an Ldata table.

Sometimes it is necessary to create a data table with an known format for its values. For example all values will occupy 4 bytes of code space even though the value itself would only occupy 1 or 2 bytes. I use the name **Byte** loosely, as 14-bit core devices use 14-bit **Words**, as opposed to 18F devices that do actually use **Bytes**.

```
Ldata 100000, 10000, 1000, 100, 10, 1
```

The above line of code would produce an uneven code space usage, as each value requires a different amount of code space to hold the values. 100000 would require 4 bytes of code space, 10000 and 1000 would require 2 bytes, but 100, 10, and 1 would only require 1 byte.

Reading these values using **Lread** would cause problems because there is no way of knowing the amount of bytes to read in order to increment to the next valid value.

The answer is to use formatters to ensure that a value occupies a predetermined amount of bytes. These are: -

Byte
Word
Dword
Float

Placing one of these formatters before the value in question will force a given length.

```
Ldata Dword 100000, Dword 10000, Dword 1000 ,_  
      Dword 100, Dword 10, Dword 1
```

Byte will force the value to occupy one byte of code space, regardless of it's value. Any values above 255 will be truncated to the least significant byte.

Word will force the value to occupy 2 bytes of code space, regardless of its value. Any values above 65535 will be truncated to the two least significant bytes. Any value below 255 will be padded to bring the memory count to 2 bytes.

Dword will force the value to occupy 4 bytes of code space, regardless of its value. Any value below 65535 will be padded to bring the memory count to 4 bytes. The line of code shown above uses the **Dword** formatter to ensure all the values in the **Ldata** table occupy 4 bytes of code space.

Float will force a value to its floating point equivalent, which always takes up 4 bytes of code space.

If all the values in an **Ldata** table are required to occupy the same amount of bytes, then a single formatter will ensure that this happens.

```
Ldata as Dword 100000, 10000, 1000, 100, 10, 1
```

The above line has the same effect as the formatter previous example using separate **Dword** formatters, in that all values will occupy 4 bytes, regardless of their value. All four formatters can be used with the **as** keyword.

The example below illustrates the formatters in use.

```
' Convert a Dword value into a string array using only BASIC commands  
' Similar principle to the Str$ command
```

```
Include "Proton_4.Inc"
```

```
Dim P10 as Dword ' Power of 10 variable  
Dim Cnt as Byte  
Dim J as Byte  
Dim Value as Byte ' Value to convert  
Dim String1[11] as Byte ' Holds the converted value  
Dim Ptr as Byte ' Pointer within the Byte array  
DelayMs 100 ' Wait for the LCD to stabilise  
Cls ' Clear the LCD  
Clear ' Clear all RAM before we start  
Value = 1234576 ' Value to convert  
Gosub DwordToStr ' Convert Value to string  
Print Str String1 ' Display the result  
Stop
```

```
-----  
' Convert a Dword value into a string array. Value to convert is placed in  
' Value  
' Byte array 'String1' is built up with the ASCII equivalent
```

```
DwordToStr:
```

```
Ptr = 0  
J = 0  
Repeat  
P10 = Lread DwordTbl + (J * 4)  
Cnt = 0  
While Value >= P10  
Value = Value - P10  
Inc Cnt  
Wend  
If Cnt <> 0 Then  
String1[Ptr] = Cnt + "0"  
Inc Ptr  
EndIf  
Inc J
```

```
Until J > 8
String1[Ptr] = Value + "0"
Inc Ptr
String1[Ptr] = 0      ' Add the null to terminate the string
Return
' Ldata table is formatted for all 32 bit values.
' Which means each value will require 4 bytes of code space
Dword_TBL:
  Ldata as Dword 1000000000, 100000000, 10000000, 1000000, 100000, 10000, _
                1000, 100, 10
```

Label names as pointers.

If a label's name is used in the list of values in an **Ldata** table, the label's address will be used. This is useful for accessing other tables of data using their address from a lookup table. See example below.

```
' Display text from two Ldata tables
' Based on their address located in a separate table

Include "Proton_4.Inc"      ' Use a 14-bit core device
Dim Address as Word
Dim DataByte as Byte
DelayMs 100                 ' Wait for the LCD to stabilise
Cls                          ' Clear the LCD
Address = Lread AddrTable   ' Locate the address of the first string
While                        ' Create an infinite loop
  DataByte = Lread Address   ' Read each character from the Ldata string
  If DataByte = 0 Then ExitLoop ' Exit if null found
  Print DataByte            ' Display the character
  Inc Address                ' Next character
Wend                          ' Close the loop
ExitLoop:
  Cursor 2,1                 ' Point to line 2 of the LCD
  Address = Lread AddrTable + 2 ' Locate the address of the second string
  While                        ' Create an infinite loop
    DataByte = Lread Address   ' Read each character from the Ldata string
    If DataByte = 0 Then ExitLoop2 ' Exit if null found
    Print DataByte            ' Display the character
    Inc Address                ' Next character
  Wend                          ' Close the loop
ExitLoop2:
  Stop

AddrTable:                  ' Table of address's
  Ldata as Word String1, String2
String1:
  Ldata "HELLO",0
String2:
  Ldata "WORLD",0
```

See also : Cdata, Cread, Data, Edata, Lread, Read, Restore.

Len

Syntax

Variable = **Len**(*Source String*)

Overview

Find the length of a **String**. (not including the null terminator) .

Operators

Variable is a user defined variable of type **Bit**, **Byte**, **Word**, **Dword**, **Float** or **Array**.

Source String can be a **String** variable, or a Quoted String of Characters. The *Source String* can also be a **Byte**, **Word**, **Float** or **Array** variable, in which case the value contained within the variable is used as a pointer to the start of the Source String's address in RAM. A third possibility for *Source String* is a label name, in which case a null terminated Quoted String of Characters is read from a **Cdata** table.

Example 1

```
' Display the length of SourceString
Device = 18F452           ' A suitable device for Strings
Declare Xtal = 20
Dim SourceString as String * 20 ' Create a String capable of 20 characters
Dim Length as Byte

SourceString = "HELLO WORLD" ' Load the source string with characters
Length = Len(SourceString)   ' Find the length
Print Dec Length             ' Display the result, which will be 11
Stop
```

Example 2

```
' Display the length of a Quoted Character String
Device = 18F452           ' A suitable device for Strings
Declare Xtal = 20

Dim Length as Byte

Length = Len("HELLO WORLD") ' Find the length
Print Dec Length             ' Display the result, which will be 11
Stop
```

Example 3

```
' Display the length of SourceString using a pointer to SourceString
Device = 18F452           ' A suitable device for Strings
Declare Xtal = 20

Dim SourceString as String * 20 ' Create a String capable of 20 characters
Dim Length as Byte              ' Display the length of SourceString
Dim SourceString as String * 20 ' Create a String capable of 20 characters
' Create a Word variable to hold the address of SourceString
Dim StringAddr as Word

SourceString = "HELLO WORLD"   ' Load the source string with characters
' Locate the start address of SourceString in RAM
StringAddr = AddressOf(SourceString)
Length = Len(StringAddr)       ' Find the length
Print Dec Length               ' Display the result, which will be 11
Stop
```

Example 4

```
' Display the length of a Cdata string
Device = 18F452      ' A suitable device for Strings
Declare Xtal = 20

Dim Length as Byte

Length = Len(Source) ' Find the length
Print Dec Length     ' Display the result, which will be 11
Stop
' Create a null terminated string of characters in code memory
Source:
Cdata "HELLO WORLD", 0
```

See also : [Creating and using Strings](#), [Creating and using Virtual Strings with Cdata](#), [Cdata](#), [Left\\$](#), [Mid\\$](#), [Right\\$](#), [Str\\$](#), [ToLower](#), [ToUpper](#), [AddressOf](#).

Left\$

Syntax

Destination String = **Left\$** (*Source String*, *Amount of characters*)

Overview

Extract *n* amount of characters from the left of a source string and copy them into a destination string.

Operators

Destination String can only be a **String** variable, and should be large enough to hold the correct amount of characters extracted from the *Source String*.

Source String can be a **String** variable, or a Quoted String of Characters. See below for more variable types that can be used for *Source String*.

Amount of characters can be any valid variable type, expression or constant value, that signifies the amount of characters to extract from the left of the *Source String*. Values start at 1 for the leftmost part of the string and should not exceed 255 which is the maximum allowable length of a String variable.

Example 1.

```
' Copy 5 characters from the left of SourceString into DestString

Device = 18F452                ' A suitable device for Strings
Dim SourceString as String * 20 ' Create a String capable of 20 characters
Dim DestString as String * 20  ' Create another String for 20 characters

SourceString = "HELLO WORLD"   ' Load the source string with characters
' Copy 5 characters from the source string into the destination string
DestString = Left$(SourceString, 5)
Print DestString               ' Display the result, which will be "HELLO"
Stop
```

Example 2.

```
' Copy 5 chars from the left of a Quoted Character String into DestString

Device = 18F452                ' A suitable device for Strings
Dim DestString as String * 20  ' Create a String capable of 20 characters

' Copy 5 characters from the quoted string into the destination string
DestString = Left$("HELLO WORLD", 5)
Print DestString               ' Display the result, which will be "HELLO"
Stop
```

The *Source String* can also be a **Byte**, **Word**, **Dword**, **Float** or **Array** variable, in which case the value contained within the variable is used as a pointer to the start of the Source String's address in RAM.

Example 3.

```
' Copy 5 characters from the left of SourceString into DestString using a
' pointer to SourceString

Device = 18F452           ' A suitable device for Strings
Dim SourceString as String * 20 ' Create a String capable of 20 characters
Dim DestString as String * 20  ' Create another String for 20 characters
' Create a Word variable to hold the address of SourceString
Dim StringAddr as Word

SourceString = "HELLO WORLD" ' Load the source string with characters
' Locate the start address of SourceString in RAM
StringAddr = AddressOf(SourceString)
' Copy 5 characters from the source string into the destination string
DestString = Left$(StringAddr, 5)
Print DestString           ' Display the result, which will be "HELLO"
Stop
```

A third possibility for *Source String* is a label name, in which case a null terminated Quoted String of Characters is read from a **Cdata** table.

Example 4.

```
' Copy 5 characters from the left of a Cdata table into DestString

Device = 18F452           ' A suitable device for Strings
Dim DestString as String * 20 ' Create a String capable of 20 characters

' Copy 5 characters from label Source into the destination string
DestString = Left$(Source, 5)
Print DestString           ' Display the result, which will be "HELLO"
Stop

' Create a null terminated string of characters in code memory
Source:
Cdata "HELLO WORLD", 0
```

See also : **Creating and using Strings, Creating and using Virtual Strings with Cdata, Cdata, Len, Mid\$, Right\$, Str\$, ToLower, ToUpper , AddressOf.**

Line

Syntax

Line *Set_Clear, Xpos Start, Ypos Start, Xpos End, Ypos End*

Overview

Draw a straight line in any direction on a graphic LCD.

Operators

Set_Clear may be a constant or variable that determines if the line will set or clear the pixels. A value of 1 will set the pixels and draw a line, while a value of 0 will clear any pixels and erase a line.

Xpos Start may be a constant or variable that holds the X position for the start of the line. Can be a value from 0 to 127.

Ypos Start may be a constant or variable that holds the Y position for the start of the line. Can be a value from 0 to 63.

Xpos End may be a constant or variable that holds the X position for the end of the line. Can be a value from 0 to 127.

Ypos End may be a constant or variable that holds the Y position for the end of the line. Can be a value from 0 to 63.

Example

```
' Draw a line from 0,0 to 120,34

Include "Proton_G4.INT"

Dim Xpos_Start as Byte
Dim Xpos_End as Byte
Dim Ypos_Start as Byte
Dim Ypos_End as Byte
Dim SetClr as Byte

DelayMs 100           ' Wait for the LCD to stabilise
Cls                   ' Clear the LCD
Xpos_Start = 0
Ypos_Start = 0
Xpos_End = 120
Ypos_End = 34
SetClr = 1
Line SetClr, Xpos_Start, Ypos_Start, Xpos_End, Ypos_End
Stop
```

See Also : **Box, Circle.**

LineTo

Syntax

LineTo *Set_Clear, Xpos End, Ypos End*

Overview

Draw a straight line in any direction on a graphic LCD, starting from the previous **Line** command's end position.

Operators

Set_Clear may be a constant or variable that determines if the line will set or clear the pixels. A value of 1 will set the pixels and draw a line, while a value of 0 will clear any pixels and erase a line.

Xpos End may be a constant or variable that holds the X position for the end of the line. Can be a value from 0 to 127.

Ypos End may be a constant or variable that holds the Y position for the end of the line. Can be a value from 0 to 63.

Example

' Draw a line from 0,0 to 120,34. Then from 120,34 to 0,63

```
Include "Proton_G4.INT"
```

```
Dim Xpos_Start as Byte
```

```
Dim Xpos_End as Byte
```

```
Dim Ypos_Start as Byte
```

```
Dim Ypos_End as Byte
```

```
Dim SetClr as Byte
```

```
DelayMs 100           ' Wait for the LCD to stabilise
```

```
Cls                   ' Clear the LCD
```

```
Xpos_Start = 0
```

```
Ypos_Start = 0
```

```
Xpos_End = 120
```

```
Ypos_End = 34
```

```
SetClr = 1
```

```
Line SetClr, Xpos_Start, Ypos_Start, Xpos_End, Ypos_End
```

```
Xpos_End = 0
```

```
Ypos_End = 63
```

```
LineTo SetClr, Xpos_End, Ypos_End
```

```
Stop
```

Notes

The **LineTo** command uses the compiler's internal system variables to obtain the end position of a previous **Line** command. These X and Y coordinates are then used as the starting X and Y coordinates of the **LineTo** command.

See Also : **Line, Box, Circle.**

LoadBit

Syntax

LoadBit *Variable, Index, Value*

Overview

Clear, or Set a bit of a variable or register using a variable index to point to the bit of interest.

Operators

Variable is a user defined variable, of type **Byte**, **Word**, or **Dword**.

Index is a constant, variable, or expression that points to the bit within *Variable* that requires accessing.

Value is a constant, variable, or expression that will be placed into the bit of interest. Values greater than 1 will set the bit.

Example

```
' Copy variable ExVar bit by bit into variable PT_Var
Device = 16F877
Declare Xtal = 4
Dim ExVar as Word
Dim Index as Byte
Dim Value as Byte
Dim PT_Var as Word
Again:
PT_Var = %0000000000000000
ExVar = %1011011000110111
Cls
For Index = 0 to 15          ' Create a loop for 16 bits
    Value = GetBit ExVar, Index ' Examine each bit of variable ExVar
    LoadBit PT_Var, Index, Value ' Set or Clear each bit of PT_Var
    Print At 1,1,Bin16 ExVar    ' Display the original variable
    Print At 2,1,Bin16 PT_Var  ' Display the copied variable
    DelayMs 100                ' Slow things down to see what's happening
Next                          ' Close the loop
Goto Again                   ' Do it forever
```

Notes

There are many ways to clear or set a bit within a variable, however, each method requires a certain amount of manipulation, either with rotates, or alternatively, the use of indirect addressing using the FSR, and INDF registers. Each method has its merits, but requires a certain amount of knowledge to accomplish the task correctly. The **LoadBit** command makes this task extremely simple by taking advantage of the indirect method using FSR, and INDF, however, this is not necessarily the quickest method, or the smallest, but it is the easiest. For speed and size optimisation, there is no shortcut to experience.

To Clear a known constant bit of a variable or register, then access the bit directly using Port.n.
i.e. PORTA.1 = 0

To Set a known constant bit of a variable or register, then access the bit directly using Port.n.
i.e. PORTA.1 = 1

If a Port is targeted by **LoadBit**, the Tris register is **not** affected.

See also : **ClearBit, GetBit, SetBit.**

LookDown

Syntax

Variable = **LookDown** *Index*, [*Constant* {, *Constant*...etc }]

Overview

Search *constants*(s) for *index* value. If *index* matches one of the *constants*, then store the matching *constant*'s position (0-N) in *variable*. If no match is found, then the *variable* is unaffected.

Operators

Variable is a user define variable that holds the result of the search.

Index is the variable/constant being sought.

Constant(s),... is a list of values. A maximum of 255 values may be placed between the square brackets, 256 if using an 18F device.

Example

```
Dim Value as Byte
Dim Result as Byte
Value = 177          ' The value to look for in the list
Result = 255         ' Default to value 255
Result = LookDown Value, [75,177,35,1,8,29,245]
Print "Value matches ", Dec Result, " in list"
```

In the above example, **Print** displays, "Value matches 1 in list" because Value (177) matches item 1 of [75,177,35,1,8,29,245]. Note that index numbers count up from 0, not 1; that is in the list [75,177,35,1,8,29,245], 75 is item 0.

If the value is not in the list, then Result is unchanged.

Notes

LookDown is similar to the index of a book. You search for a topic and the index gives you the page number. Lookdown searches for a value in a list, and stores the item number of the first match in a variable.

LookDown also supports text phrases, which are basically lists of byte values, so they are also eligible for Lookdown searches:

```
Dim Value as Byte
Dim Result as Byte
Value = 101          ' ASCII "e". the value to look for in the list
Result = 255         ' Default to value 255
Result = LookDown Value, ["Hello World"]
```

In the above example, Result will hold a value of 1, which is the position of character 'e'

See also : **Cdata, Cread, Data, Edata, Eread, Ldata, LookDownL, LookUp, LookUpL, Lread, Read, Restore.**

LookDownL

Syntax

Variable = **LookDownL** *Index*, {*Operator*} [*Value* {, *Value*...etc }]

Overview

A comparison is made between *index* and *value*; if the result is true, 0 is written into *variable*. If that comparison was false, another comparison is made between *value* and *value1*; if the result is true, 1 is written into *variable*. This process continues until a true is yielded, at which time the *index* is written into *variable*, or until all entries are exhausted, in which case *variable* is unaffected.

Operators

Variable is a user define variable that holds the result of the search.

Index is the variable/constant being sought.

Value(s) can be a mixture of 16-bit constants, string constants and variables. Expressions may not be used in the *Value* list, although they may be used as the *index* value. A maximum of 85 values may be placed between the square brackets, 256 if using an 18F device.

Operator is an optional comparison operator and may be one of the following: -

- = equal
- <> not equal
- > greater than
- < less than
- >= greater than or equal to
- <= less than or equal to

The optional operator can be used to perform a test for other than equal to ("=") while searching the list. For example, the list could be searched for the first *Value* greater than the *index* parameter by using ">" as the *operator*. If *operator* is left out, "=" is assumed.

Example

```
Var1 = LookDownL Wrd, [ 512, Wrd1, 1024 ]  
Var1 = LookDownL Wrd, < [ 10, 100, 1000 ]
```

Notes

Because **LookDownL** is more versatile than the standard **LookDown** command, it generates larger code. Therefore, if the search list is made up only of 8-bit constants and strings, use **LookDown**.

See also : **Cdata, Cread, Cread8, Cread16, Cread32, Edata, Eread, Ldata, LookDown, LookUp, LookUpL, Lread, Lread8, Lread16, Lread32.**

LookUp

Syntax

Variable = **LookUp** *Index*, [*Constant* {, *Constant*...etc }]

Overview

Look up the value specified by the index and store it in variable. If the index exceeds the highest index value of the items in the list, then variable remains unchanged.

Operators

Variable may be a constant, variable, or expression. This is where the retrieved value will be stored.

Index may be a constant or variable. This is the item number of the value to be retrieved from the list.

Constant(s) may be any 8-bit value (0-255). A maximum of 255 values may be placed between the square brackets, 256 if using an 18F device.

Example

```
' Create an animation of a spinning line.
Dim Index as Byte
Dim Frame as Byte
Cls                               ' Clear the LCD
Rotate:
  For Index = 0 to 3              ' Create a loop of 4
    Frame = LookUp Index, [ "|\"-/\" ] ' Table of animation characters
    Print At 1, 1, Frame         ' Display the character
    DelayMs 200                  ' So we can see the animation
  Next                            ' Close the loop
  Goto Rotate                     ' Repeat forever
```

Notes

index starts at value 0. For example, in the **LookUp** command below. If the first value (10) is required, then index will be loaded with 0, and 1 for the second value (20) etc.

```
Var1 = LookUp Index, [10, 20, 30]
```

See also : Cdata, Cread, Cread8, Cread16, Cread32, Edata, Eread, Ldata, LookDown, LookDownL, LookUpL, Lread, Lread8, Lread8, Lread32.

LookUpL

Syntax

Variable = **LookUpL** *Index*, [*Value* {, *Value*...etc }]

Overview

Look up the value specified by the index and store it in variable. If the index exceeds the highest index value of the items in the list, then variable remains unchanged. Works exactly the same as **LookUp**, but allows variable types or constants in the list of values.

Operators

Variable may be a constant, variable, or expression. This is where the retrieved value will be stored.

Index may be a constant or variable. This is the item number of the value to be retrieved from the list.

Value(s) can be a mixture of 16-bit constants, string constants and variables. A maximum of 85 values may be placed between the square brackets, 256 if using an 18F device.

Example

```
Dim Var1 as Byte
Dim Wrd as Word
Dim Index as Byte
Dim Assign as Word
Var1 = 10
Wrd = 1234
Index = 0      ' Point to the first value in the list (Wrd)
Assign = LookUpL Index, [Wrd, Var1, 12345]
```

Notes

Expressions may not be used in the *Value* list, although they may be used as the *Index* value.

Because **LookUpL** is capable of processing any variable and constant type, the code produced is a lot larger than that of **LookUp**. Therefore, if only 8-bit constants are required in the list, use **LookUp** instead.

See also : **Cdata, Cread, Cread8, Cread16, Cread32 Edata, Eread, Ldata, LookDown, LookDownL, LookUp, Lread, Lread8, Lread16, Lread32.**

Low

Syntax

Low *Port* or *Port.Bit*

Overview

Place a Port or bit in a low state. For a port, this means filling it with 0's. For a bit this means setting it to 0.

Operators

Port can be any valid port.

Port.Bit can be any valid port and bit combination, i.e. PORTA.1

Example

```
Symbol LED = PORTB.4
Low LED
Low PORTB.0      ' Clear PORTB bit 0
Low PORTB        ' Clear all of PORTB
```

See also : Dim, High, Symbol.

Lread

Syntax

Variable = **Lread** *Label*

Overview

Read a value from an **Ldata** table and place into *Variable*

Operators

Variable is a user defined variable.

Label is a label name preceding the **Ldata** statement, or expression containing the *Label* name.

Example

```
Device = 16F877
Dim Char as Byte
Dim Loop as Byte
Cls
For Loop = 0 to 9           ' Create a loop of 10
    Char = Lread Label + Loop ' Read memory location Label + Loop
    Print Char              ' Display the value read
Next
Stop
Label: Ldata "HELLO WORLD" ' Create a string of text in code memory
```

The program above reads and displays 10 values from the address located by the *Label* accompanying the **Ldata** command. Resulting in "HELLO WORL" being displayed.

Ldata is not simply used for character storage, it may also hold 8, 16, 32 bit, or floating point values. The example below illustrates this: -

```
Device = 16F628
Dim Var1 as Byte
Dim Wrld as Word
Dim Dwd1 as Dword
Dim Flt1 as Float
Cls
Var1 = Lread Bit8_Val      ' Read the 8-bit value
Print Dec Var1, " "
Wrld = Lread Bit16_Val     ' Read the 16-bit value
Print Dec Wrld
Dwd1 = Lread Bit32_Val     ' Read the 32-bit value
Print At 2,1, Dec Dwd1, " "
Flt1 = Lread Flt_Val      ' Read the floating point value
Print Dec Flt1
Stop
Bit8_Val: Ldata 123
Bit16_Val: Ldata 1234
Bit32_Val: Ldata 123456
Flt_Val: Ldata 123.456
```

Floating point examples.

14-bit core example

```
' 14-bit read floating point data from a table and display the results
Device = 16F877
Dim Flt as Float           ' Declare a Floating Point variable
Dim Fcount as Byte
Cls                         ' Clear the LCD
Fcount = 0                 ' Clear the table counter
Repeat                     ' Create a loop
  Flt = Lread FlTable + Fcount ' Read the data from the Ldata table
  Print At 1, 1, Dec3 Flt    ' Display the data read
  Fcount = Fcount + 4      ' Point to next value, by adding 4 to counter
  DelayMs 1000             ' Slow things down
Until Flt = 0.005         ' Stop when 0.005 is read
Stop

FlTable:
  Ldata as Float 3.14, 65535.123, 1234.5678, -1243.456, -3.14, 998999.12, _
                0.005
```

18F device example

```
' Read floating point data from a table and display the results
Device = 18F452
Dim Flt as Float           ' Declare a Floating Point variable
Dim Fcount as Byte
Cls                         ' Clear the LCD
Fcount = 0                 ' Clear the table counter
Repeat                     ' Create a loop
  Flt = Lread FlTable + Fcount ' Read the data from the Ldata table
  Print At 1, 1, Dec3 Flt    ' Display the data read
  Fcount = Fcount + 2      ' Point to next value, by adding 2 to counter
  DelayMs 1000             ' Slow things down
Until Flt = 0.005         ' Stop when 0.005 is read
Stop

FlTable:
  Ldata as Float 3.14, 65535.123, 1234.5678, -1243.456, -3.14, 998999.12, _
                0.005
```

Notes

Ldata tables should be placed at the end of the BASIC program. If an **Ldata** table is placed at the beginning of the program, then a **Goto** command must jump over the tables, to the main body of code.

```
Goto OverDataTable
  Ldata 1,2,3,4,5,6
OverDataTable:

  { rest of code here }
```

With 14-bit core devices, an 8-bit value (0 - 255) in an **Ldata** statement will occupy a single code space, however, 16-bit data (0 - 65535) will occupy two spaces, 32-bit and floating point values will occupy 4 spaces. This must be taken into account when using the **Lread** command. See 14-bit floating point example above.

With **18F** devices, an 8, and 16-bit value in an **Ldata** statement will occupy a single code space, however, 32-bit and floating point values will occupy 2 spaces. This must be taken into account when using the **Lread** command. See previous 16-bit floating point example.

See also : **Cdata, Cread, Cread8, Cread16, Cread32, Ldata.**

Lread8, Lread16, Lread32

Syntax

Variable = **Lread8** *Label* [*Offset Variable*]

or

Variable = **Lread16** *Label* [*Offset Variable*]

or

Variable = **Lread32** *Label* [*Offset Variable*]

Overview

Read an 8, 16, or 32-bit value from an **Ldata** table using an offset of *Offset Variable* and place into *Variable*, with more efficiency than using **Lread**. For PICmicro's that can access their own code memory, such as the 16F87x and all the 18F range.

Lread8 will access 8-bit values from an **Ldata** table.

Lread16 will access 16-bit values from an **Ldata** table.

Lread32 will access 32-bit values from an **Ldata** table, this also includes floating point values.

Operators

Variable is a user defined variable of type **Bit**, **Byte**, **Word**, **Dword**, **Float** or **Array**.

Label is a label name preceding the **Ldata** statement of which values will be read from.

Offset Variable can be a constant value, variable, or expression that points to the location of interest within the **Ldata** table.

Lread8 Example

```
' Extract the second value from within an 8-bit Ldata table
Device = 16F877
Dim Offset as Byte      ' Declare a Byte size variable for the offset
Dim Result as Byte      ' Declare a Byte size variable to hold the result

Cls                      ' Clear the LCD
Offset = 1               ' Point to the second value in the Ldata table
' Read the 8-bit value pointed to by Offset
Result = Lread8 Byte_Table[Offset]
Print Dec Result        ' Display the decimal result on the LCD
Stop

' Create a table containing only 8-bit values
Byte_Table: Ldata as Byte 100, 200
```

Lread16 Example

```
' Extract the second value from within a 16-bit Ldata table
Device = 16F877
Dim Offset as Byte      ' Declare a Byte size variable for the offset
Dim Result as Word     ' Declare a Word size variable to hold the result

Cls                    ' Clear the LCD
Offset = 1             ' Point to the second value in the Ldata table
' Read the 16-bit value pointed to by Offset
Result = Lread16 WordTable[Offset]
Print Dec Result      ' Display the decimal result on the LCD
Stop

' Create a table containing only 16-bit values
WordTable: Ldata as Word 1234, 5678
```

Lread32 Example

```
' Extract the second value from within a 32-bit Ldata table
Device = 16F877
Dim Offset as Byte      ' Declare a Byte size variable for the offset
Dim Result as Dword    ' Declare a Dword size variable to hold the result

Cls                    ' Clear the LCD
Offset = 1             ' Point to the second value in the Ldata table
' Read the 32-bit value pointed to by Offset
Result = Lread32 DwordTable[Offset]
Print Dec Result      ' Display the decimal result on the LCD
Stop

' Create a table containing only 32-bit values
DwordTable: Ldata as Dword 12340, 56780
```

Notes

Data storage in any program is of paramount importance, and although the standard **Lread** command can access multi-byte values from an **Ldata** table, it was not originally intended as such, and is more suited to accessing character data or single 8-bit values. However, the **Lread8**, **Lread16**, and **Lread32** commands are specifically written in order to efficiently read data from an **Ldata** table, and use the least amount of code space in doing so, thus increasing the speed of operation. Which means that wherever possible, **Lread** should be replaced by **Lread8**, **Lread16**, or **Lread32**.

See also :**Cdata**, **Cread**, **Cread8**, **Cread16**, **Cread32**, **Ldata**, **Lread**.

Mid\$

Syntax

Destination String = **Mid\$** (*Source String*, *Position within String*, *Amount of characters*)

Overview

Extract *n* amount of characters from a source string beginning at *n* characters from the left, and copy them into a destination string.

Operators

Destination String can only be a **String** variable, and should be large enough to hold the correct amount of characters extracted from the *Source String*.

Source String can be a **String** variable, or a **Quoted String of Characters**. See below for more variable types that can be used for *Source String*.

Position within String can be any valid variable type, expression or constant value, that signifies the position within the *Source String* from which to start extracting characters. Values start at 1 for the leftmost part of the string and should not exceed 255 which is the maximum allowable length of a *String* variable.

Amount of characters can be any valid variable type, expression or constant value, that signifies the amount of characters to extract from the left of the *Source String*. Values start at 1 and should not exceed 255 which is the maximum allowable length of a *String* variable.

Example 1

```
' Copy 5 characters from position 4 of SourceString into DestString

Device = 18F452           ' A suitable device for Strings
Dim SourceString as String * 20 ' Create a String of 20 characters
Dim DestString as String * 20  ' Create another String

SourceString = "HELLO WORLD" ' Load the source string with characters
' Copy 5 characters from the source string into the destination string
DestString = Mid$(SourceString, 4, 5)
Print DestString          ' Display the result, which will be "LO WO"
Stop
```

Example 2

```
' Copy 5 chars from position 4 of a Quoted Character String into DestString

Device = 18F452           ' A suitable device for Strings
Dim DestString as String * 20 ' Create a String of 20 characters

' Copy 5 characters from the quoted string into the destination string
DestString = Mid$("HELLO WORLD", 4, 5)
Print DestString          ' Display the result, which will be "LO WO"
Stop
```

The *Source String* can also be a **Byte**, **Word**, **Dword**, **Float** or **Array** variable, in which case the value contained within the variable is used as a pointer to the start of the *Source String*'s address in RAM.

Example 3

```
' Copy 5 chars from position 4 of SourceString to DestString with a pointer
' to SourceString

Device = 18F452          ' A suitable device for Strings
Dim SourceString as String * 20 ' Create a String of 20 characters
Dim DestString as String * 20  ' Create another String
' Create a Word variable to hold the address of SourceString
Dim StringAddr as Word

SourceString = "HELLO WORLD" ' Load the source string with characters
' Locate the start address of SourceString in RAM
StringAddr = AddressOf(SourceString)
' Copy 5 characters from the source string into the destination string
DestString = Mid$(StringAddr, 4, 5)
Print DestString          ' Display the result, which will be "LO WO"
Stop
```

A third possibility for *Source String* is a Label name, in which case a null terminated Quoted String of Characters is read from a **Cdata** table.

Example 4

```
' Copy 5 characters from position 4 of a Cdata table into DestString

Device = 18F452          ' A suitable device for Strings
Dim DestString as String * 20 ' Create a String of 20 characters

' Copy 5 characters from label Source into the destination string
DestString = Mid$(Source, 4, 5)
Print DestString          ' Display the result, which will be "LO WO"
Stop

' Create a null terminated string of characters in code memory
Source:
Cdata "HELLO WORLD", 0
```

See also : **Creating and using Strings, Creating and using Virtual Strings with Cdata, Cdata, Len, Left\$, Right\$, Str\$, ToLower, ToUpper, AddressOf.**

On Goto

Syntax

On *Index Variable* **Goto** *Label1* {...*Labeln*}

Overview

Cause the program to jump to different locations based on a variable index. On a PICmicro™ device with only one page of memory. Exactly the same functionality as **Branch**.

Operators

Index Variable is a constant, variable, or expression, that specifies the label to jump to.

Label1...Labeln are valid labels that specify where to branch to. A maximum of 255 labels may be placed after the **Goto**, 256 if using an 18F device.

Example

```
Device = 16F84
Dim Index as Byte

Cls                               ' Clear the LCD
Index = 2                          ' Assign Index a value of 2
Start:                             ' Jump to label 2 (Label_2) because Index = 2
  On Index Goto Label_0, Label_1, Label_2

Label_0:
  Index = 2                          ' Index now equals 2
  Print At 1,1,"Label 0"             ' Display the Label name on the LCD
  DelayMs 500                        ' Wait 500ms
  Goto Start                          ' Jump back to Start

Label_1:
  Index = 0                          ' Index now equals 0
  Print At 1,1,"Label 1"             ' Display the Label name on the LCD
  DelayMs 500                        ' Wait 500ms
  Goto Start                          ' Jump back to Start

Label_2:
  Index = 1                          ' Index now equals 1
  Print At 1,1,"Label 2"             ' Display the Label name on the LCD
  DelayMs 500                        ' Wait 500ms
  Goto Start                          ' Jump back to Start
```

The above example we first assign the index variable a value of 2, then we define our labels. Since the first position is considered 0 and the variable Index equals 2 the **On Goto** command will cause the program to jump to the third label in the list, which is Label_2.

Notes

On Goto is useful when you want to organise a structure such as: -

```
If Var1 = 0 Then Goto Label_0 ' Var1 = 0: go to label "Label_0"  
If Var1 = 1 Then Goto Label_1 ' Var1 = 1: go to label "Label_1"  
If Var1 = 2 Then Goto Label_2 ' Var1 = 2: go to label "Label_2"
```

You can use **On Goto** to organise this into a single statement: -

```
On Var1 Goto Label_0, Label_1, Label_2
```

This works exactly the same as the above **If...Then** example. If the value is not in range (in this case if Var1 is greater than 2), **On Goto** does nothing. The program continues with the next instruction.

The **On Goto** command is primarily for use with PICmicro™ devices that have one page of memory (0-2047). If larger PICmicros are used and you suspect that the branch label will be over a page boundary, use the **On GotoL** command instead.

See also : **Branch, BranchL, On GotoL, On Gosub.**

On GotoL

Syntax

On *Index Variable* **GotoL** *Label1* {...*LabelN*}

Overview

Cause the program to jump to different locations based on a variable index. On a PICmicro™ device with more than one page of memory, or 18F devices. Exactly the same functionality as **BranchL**.

Operators

Index Variable is a constant, variable, or expression, that specifies the label to jump to.

Label1...LabelN are valid labels that specify where to branch to. A maximum of 127 labels may be placed after the **GotoL**, 256 if using an 18F device.

Example

```
Device = 16F877           ' Use a larger PICmicro device
Dim Index as Byte

Cls                       ' Clear the LCD
Index = 2                 ' Assign Index a value of 2
Start:                    ' Jump to label 2 (Label_2) because Index = 2
  On Index GotoL Label_0, Label_1, Label_2

Label_0:
  Index = 2               ' Index now equals 2
  Print At 1,1,"Label 0" ' Display the Label name on the LCD
  DelayMs 500             ' Wait 500ms
  Goto Start              ' Jump back to Start

Label_1:
  Index = 0               ' Index now equals 0
  Print At 1,1,"Label 1" ' Display the Label name on the LCD
  DelayMs 500             ' Wait 500ms
  Goto Start              ' Jump back to Start

Label_2:
  Index = 1               ' Index now equals 1
  Print At 1,1,"Label 2" ' Display the Label name on the LCD
  DelayMs 500             ' Wait 500ms
  Goto Start              ' Jump back to Start
```

The above example we first assign the index variable a value of 2, then we define our labels. Since the first position is considered 0 and the variable Index equals 2 the **On GotoL** command will cause the program to jump to the third label in the list, which is Label_2.

Notes

The **On GotoL** command is mainly for use with PICmicro™ devices that have more than one page of memory (greater than 2048). It may also be used on any PICmicro™ device, but does produce code that is larger than **On Goto**.

See also : **Branch, BranchL, On Goto, On Gosub .**

On Gosub

Syntax

On *Index Variable* **Gosub** *Label1* {...*LabelN*}

Overview

Cause the program to Call a subroutine based on an index value. A subsequent **Return** will continue the program immediately following the **On Gosub** command.

Operators

Index Variable is a constant, variable, or expression, that specifies the label to call.

Label1...LabelN are valid labels that specify where to call. A maximum of 256 labels may be placed after the **Gosub**.

Example

```
Device = 18F452           ' Use an 18F PICmicro
Dim Index as Byte

Cls                       ' Clear the LCD
While                    ' Create an infinite loop
  For Index = 0 to 2     ' Create a loop to call all the labels
    ' Call the label depending on the value of Index
    On Index Gosub Label_0, Label_1, Label_2
    DelayMs 500         ' Wait 500ms after the subroutine has returned
  Next
Wend                      ' Do it forever
Label_0:
  Print At 1,1,"Label 0" ' Display the Label name on the LCD
  Return
Label_1:
  Print At 1,1,"Label 1" ' Display the Label name on the LCD
  Return
Label_2:
  Print At 1,1,"Label 2" ' Display the Label name on the LCD
  Return
```

The above example, a loop is formed that will load the variable Index with values 0 to 2. The **On Gosub** command will then use that value to call each subroutine in turn. Each subroutine will **Return** to the **DelayMs** command, ready for the next scan of the loop.

Notes

On Gosub is useful when you want to organise a structure such as: -

```
If Var1 = 0 Then Gosub Label_0 ' Var1 = 0: call label "Label_0"  
If Var1 = 1 Then Gosub Label_1 ' Var1 = 1: call label "Label_1"  
If Var1 = 2 Then Gosub Label_2 ' Var1 = 2: call label "Label_2"
```

You can use **On Gosub** to organise this into a single statement: -

```
On Var1 Gosub Label_0, Label_1, Label_2
```

This works exactly the same as the above **If...Then** example. If the value is not in range (in this case if Var1 is greater than 2), **On Gosub** does nothing. The program continues with the next instruction..

On Gosub is only supported with 18F devices because they are the only PICmicro™ devices that allow code access to their return stack, which is required for the computed **Return** address.

See also : Branch, BranchL, On Goto, On GotoL.

On_Hardware_Interrupt

Syntax

On_Hardware_Interrupt *Label*

Overview

Point to the subroutine that will be called when a hardware interrupt occurs. High priority hardware interrupt if using an 18F device.

Operators

Label is a valid identifier

Example

```
' Flash an LED on PORTB.0 at a different rate to the LED on PORTB.1
'
Device = 16F877
Declare Xtal = 4
On_Hardware_Interrupt Goto ISR_Flash

Goto Main          ' Jump over the interrupt subroutine
' -----
' Timer0 overflow interrupt handler starts here
' Xor PORTB with 1, which will turn on the LED connected to PORTB.0
' with one interrupt and turn it off with the next interrupt
'
ISR_Flash:
Context Save      ' Save any variables or SFRs before the interrupt starts
If INTCONbits_T0IF = 1 Then ' Is it TMR0 that caused the interrupt?
    PORTB = PORTB ^ 1      ' Yes. So. Xor PORTB.0
    INTCONbits_T0IF = 0    ' Clear the TMR0 overflow flag
EndIf
Context Restore  ' Restore any variables or SFRs and exit the interrupt
' -----
' The main program loop starts here
'
Main:
Low PORTB = 0      ' Make PORTB all outputs and pull it low
'
' Initiate the interrupt
'
OPTION_REG = %00101111 ' Setup Timer0
TMR0 = 0             ' Clear TMR0 initially
INTCONbits_T0IE = 1  ' Enable a Timer0 overflow interrupt
INTCONbits_GIE = 1   ' Enable global interrupts
While               ' Create an infinite loop
    Clear PORTB.1    ' Extinguish the LED
    DelayMs 500      ' Wait a while
    Set PORTB.1      ' Illuminate the LED
    DelayMs 500      ' Wait a while
Wend
```

Typical format of the interrupt handler with standard 14-bit core devices.

The interrupt handler subroutine must always follow a fixed pattern.

- First, the contents of the **STATUS**, **PCLATH**, and Working Register (**WREG**) must be saved, this is termed *context saving*, and is performed when the command **Context Save** is issued. Variable space is automatically allocated for these registers in the shared portion of memory located at the top of RAM Bank 0. The Context Save command also instructs the compiler to save any compiler system variables and device SFRs (Special Function Registers) used within the interrupt handler. Note that "within the interrupt handler" means code between **Context Save** and **Context Restore**. It will **not** track any **Goto** or **Gosub** commands.
- Because a standard 14-bit core device has a single interrupt vector, the cause of the interrupt must be ascertained by checking the appropriate flag. For example **INTCON.T0IF** for a Timer0 overflow interrupt, and only perform the relevant code for the relevant interrupt. This is accomplished by a simple **If-EndIf**. For example:

ISR_Handler:

```
Context Save                ' Save any variables or SFRs used
If INTCONbits_T0IF = 1 Then ' Is it Timer0 that caused the interrupt?
    Print "Hello World"      ' Yes. So do this code
    INTCONbits_T0IF = 0      ' Clear the Timer0 overflow flag
EndIf
Context Restore           ' Restore any variables or SFRs used and exit
```

If more than one interrupt is enabled, multiple **If-EndIf** conditions will be required within the single interrupt handling subroutine.

- The previously saved **STATUS**, **PCLATH**, and Working register (**WREG**) must be returned to their original conditions (*context restoring*) once the interrupt handler has performed its task. The **Context Restore** command is used for this. It also returns the program back to the main body code where the interrupt was called from. In other words it performs an assembler **Retfie** instruction.

The above code snippet will cause several compiler system variables and device SFRs to be saved and restored, thus causing little, or no, disturbance to the main body code.

Typical format of the interrupt handler with enhanced 14-bit core devices.

As with standard 14-bit core interrupts, the interrupt handler subroutine must follow a fixed pattern.

- First, the **Context Save** command should be issued, as this will save any compiler system variables and SFRs used. The microcontroller itself will save the contents of the **STATUS**, **PCLATH**, **BSR**, **FSR0L\H**, **FSR1L\H** and **WREG** registers.
- As with standard 14-bit core devices, enhanced 14-bit core devices have a single interrupt vector, therefore the same rules apply as outlined above concerning the establishment of the cause of the interrupt. Not forgetting to clear any interrupt flag that needs clearing before exiting the interrupt.

- The **Context Restore** command should be issued at the end of the interrupt handler, as long as its corresponding **Context Save** command was used previously. This will restore any compiler system variables and SFRs, then exit the interrupt using the **Retfie** mnemonic.

Note that the **STATUS**, **PCLATH**, **BSR**, **FSR0L\H**, **FSR1L\H** and **WREG** registers will automatically be restored by the microcontroller once the interrupt is exited.

As with standard 14-bit core devices, any compiler variable or device SFR that is used by a command will be saved and restored as long as they reside within the **Context Save** and **Context Restore** commands. This is termed Managed Interrupts.

Note that the **Context Save** and **Context Restore** commands are not required unless managed interrupts are implemented, in which case use the **Retfie** mnemonic to exit the interrupt handler. However, you must be certain that the interrupt handler is not disturbing any compiler system variables or SFRs, or your program will not run correctly.

Typical format of the interrupt handler with 18F devices.

As with both types of 14-bit core devices, the interrupt handler subroutine must also follow a fixed pattern.

- First, the **Context Save** command should be issued, as this will save any compiler system variables and SFRs used. The microcontroller itself will save the contents of the **STATUS**, **BSR** and **WREG** registers for a high priority interrupt.
- 18F devices have two interrupt vectors for high and low priority interrupts, see **On_Low_Interrupt**. However, both of these must follow the rules laid down for 14-bit core devices, in that the cause of the interrupt must be ascertained before the appropriate code is performed, and any interrupt flag that needs clearing must be cleared before exiting the interrupt.
- The **Context Restore** command should be issued at the end of the interrupt handler, as long as its corresponding **Context Save** command was used previously. This will restore any compiler system variables and SFRs, then exit the interrupt using the **Retfie 1** mnemonic.

Note that the **STATUS**, **BSR** and **WREG** registers will automatically be restored by the microcontroller once the interrupt is exited from a high priority interrupt.

Upon exiting the interrupt, a simple **Retfie 1** (Return From Interrupt Fast) mnemonic can be used, as long as the **Context Save** command is not issued and it is certain that the interrupt handling subroutine is not disturbing any compiler system variables or device SFRs.

Note.

On all devices, the code within the interrupt handler should be as quick and efficient as possible because while it's processing the code, the main program is halted. When inside an interrupt, care should be taken to ensure that the watchdog timer does not time-out, if it's enabled. Placing a **ClrWdt** mnemonic at the beginning of the interrupt handler will usually prevent this from happening. An alternative approach would be to disable the watchdog timer altogether at programming time, which is the default of the compiler.

On_Low_Interrupt

Syntax

On_Low_Interrupt *Label*

Overview

Point to the subroutine that will be called when a Low Priority Hardware interrupt occurs on an 18F device.

Operators

Label is a valid identifier

Example

```
' Demonstrate the use of context saving of the compiler's System variables
' Creates low and high priority interrupts incrementing on Timer0 and Timer1
' Within the interrupts a value is displayed and incremented
' In the foreground another value is incremented and transmitted serially
'
Include "Proton18_4.Inc" ' Use the Proton Board with an 18F device
Declare Optimiser_Level = 3 ' Use max optimisation for tight/fast Asm code
' Point to the High Priority interrupt handler to the subroutine
On_Hardware_Interrupt Goto ISR_High
' Point to the Low Priority interrupt handler to the subroutine
On_Low_Interrupt Goto ISR_Low
'
' Create some variables
'
Dim HighCounter as Dword          ' Counter for the high interrupt routine
Dim LowCounter as Dword          ' Counter for the low interrupt routine
Dim ForeGroundCounter as Dword   ' Counter for the Foreground routine
Dim wTimer0 as TMR0L.Word       ' Create a 16-bit Word from registers TMR0L/H
Dim wTimer1 as TMR1L.Word       ' Create a 16-bit Word from registers TMR1L/H
'-----
Goto Main                        ' Jump over any subroutines
'
'-----
' High Priority Hardware Interrupt Handler
' Interrupt's on a Timer1 Overflow. Display on the LCD and increment a value
'
ISR_High:
'
' Save the compiler's system variables used in the interrupt routine only
' Also save any SFRs used
'
Context Save PORTD, TRISD
If PIR1bits_TMR1IF = 1 Then ' Is it a Timer1 overflow interrupt?
    ' Yes. So Display the value on the LCD
    Print at 1,1,"High Int ", Dec HighCounter
    Inc HighCounter         ' Increment the value
    PIR1bits_TMR1IF = 0    ' Clear the Timer1 Overflow flag
EndIf
'
' Restore compiler's system variables used within the interrupt routine only
' and exit the interrupt
'
Context Restore
```

```
' -----  
' Low Priority Hardware Interrupt Handler  
' Interrupt's on a Timer0 Overflow  
' Display on the LCD and increment a value  
'  
ISR_Low:  
' Save the compiler's system variables used in the interrupt routine only  
' Also save any SFR's used.  
'  
Context Save PORTD, TRISD  
If INTCON1bits_TMR0IF = 1 Then ' Is it a Timer0 overflow interrupt?  
'  
' Yes. So Disable Timer 1 High priority interrupt while we use the LCD  
'  
    PIE1bits_TMR1IE = 0 ' Display the value on line 2 of the LCD  
    Print at 2,1,"Low Int ", Dec LowCounter," "  
    Inc LowCounter ' Increment the value  
    PIE1bits_TMR1IE = 1 ' Re-Enable the Timer1 High priority interrupt  
    INTCON1bits_TMR0IF = 0 ' Clear the Timer0 Overflow flag  
EndIf  
'  
' Restore the compiler's system variables used in the interrupt routine only  
' and exit the interrupt  
'  
Context Restore  
'  
' -----  
' The Main Program Loop Starts Here  
'  
Main:  
    Delaysms 100 ' Wait for the LCD to stabilise  
    INTCON1 = 0 ' Disable Interrupts  
    Low PORTD ' Set PORTD to Output Low  
    HighCounter = 0  
    LowCounter = 0  
    ForeGroundCounter = 0  
    Cls ' Clear the LCD  
'  
' Setup Timer0  
'  
    T0CONbits_T0PS2 = 0 ' \  
    T0CONbits_T0PS1 = 0 ' Timer0 Prescaler to 1:4  
    T0CONbits_T0PS0 = 1 ' /  
    T0CONbits_PSA = 0 ' Assign the prescaler  
    T0CONbits_T0CS = 0 ' Increment on the internal Clk  
    T0CONbits_T08Bit = 0 ' Timer0 is configured as a 16-bit counter  
    wTimer0 = 0 ' Clear Timer0  
    T0CONbits_TMR0ON = 1 ' Enable Timer0  
'  
' Setup Timer1  
'  
    T1CONbits_RD16 = 1 ' Enable Timer1 in 16-bit operation  
    T1CONbits_T1CKPS1 = 0 ' \  
    T1CONbits_T1CKPS0 = 0 ' Timer1 Prescaler to 1:2  
    T1CONbits_T1OSCEN = 0 ' /  
    T1CONbits_T1OSCCEN = 0 ' Disable External Oscillator  
    T1CONbits_TMR1CS = 0 ' Increment on the internal Clk  
    wTimer1 = 0 ' Clear Timer1  
    T1CONbits_TMR1ON = 1 ' Enable Timer1
```

```
,  
' Setup the High and Low priority interrupts  
,  
INTCON1bits_TMR0IE = 1      ' Enable the Timer0 overflow interrupt  
INTCON2bits_TMR0IP = 0      ' Timer0 Overflow Interrupt to Low priority  
INTCON1bits_TMR1IE = 1      ' Enable the Timer1 overflow interrupt  
IPR1bits_TMR1IP = 1         ' Timer1 Overflow Interrupt to High priority  
RCONbits_IPEN = 1          ' Enable priority levels on interrupts  
INTCON1bits_GIEL = 1        ' Enable low priority peripheral interrupts  
INTCON1bits_GIE = 1         ' Enable all high priority interrupts  
,  
' Display value in foreground while interrupts do their thing in background  
,  
While                                ' Create an infinite loop  
    ' Display the value on serial terminal  
    Hrsout "ForeGround ", Dec ForeGroundCounter, 13  
    Inc ForeGroundCounter      ' Increment the value  
    Delays 200  
Wend                                ' Close the loop. i.e. do it forever
```

Output

Syntax

Output *Port* or *Port.Pin*

Overview

Makes the specified *Port* or *Port.Pin* an output.

Operators

Port.Pin must be a Port.Pin constant declaration.

Example

```
Output PORTA.0      ' Make bit-0 of PORTA an output
Output PORTA        ' Make all of PORTA an output
```

Notes

An Alternative method for making a particular pin an output is by directly modifying the Tris: -

```
TRISB.0 = 0          ' Set PORTB, bit-0 to an output
```

All of the pins on a port may be set to output by setting the whole Tris register at once: -

```
TRISB = %00000000   ' Set all of PORTB to outputs
```

In the above examples, setting a Tris bit to 0 makes the pin an output, and conversely, setting the bit to 1 makes the pin an input.

See also : [Input.](#)

Org

Syntax

Org *Value*

Overview

Set the program origin for subsequent code at the address defined in *Value*

Operators

Value can be any constant value within the range of the particular microcontroller's memory.

Example

Device **16F877**

```
Org 2000          ' Set the origin to address 2000
Cdata 120, 243, "Hello"  ' Place data starting at address 2000
```

OR

```
Symbol Address = 2000
```

```
Org Address + 1    ' Set the origin to address 2001
Cdata 120, 243, "Hello"  ' Place data starting at address 2001
```

Notes

If more complex values are required after the **Org** directive, such as assembler variables etc.

Use :-

```
@ Org { assembler variables etc }
```

Oread

Syntax

Oread *Pin*, *Mode*, [*Inputdata*]

Overview

Receive data from a device using the Dallas Semiconductor 1-wire protocol. The 1-wire protocol is a form of asynchronous serial communication developed by Dallas Semiconductor. It requires only one I/O pin which may be shared between multiple 1-wire devices.

Operators

Pin is a Port-Bit combination that specifies which I/O pin to use. 1-wire devices require only one I/O pin (normally called DQ) to communicate. This I/O pin will be toggled between output and input mode during the **Oread** command and will be set to input mode by the end of the **Oread** command.

Mode is a numeric constant (0 - 7) indicating the mode of data transfer. The Mode argument control's the placement of reset pulses and detection of presence pulses, as well as byte or bit input. See notes below.

Inputdata is a list of variables or arrays to store the incoming data into.

Example

```
Dim Result as Byte
Symbol DQ = PORTA.0
Oread DQ, 1, [Result]
```

The above example code will transmit a 'reset' pulse to a 1-wire device (connected to bit 0 of PORTA) and will then detect the device's 'presence' pulse and receive one byte and store it in the variable Result.

Notes

The Mode operator is used to control placement of reset pulses (and detection of presence pulses) and to designate byte or bit input. The table below shows the meaning of each of the 8 possible value combinations for Mode.

Mode Value	Effect
0	No Reset, Byte mode
1	Reset before data, Byte mode
2	Reset after data, Byte mode
3	Reset before and after data, Byte mode
4	No Reset, Bit mode
5	Reset before data, Bit mode
6	Reset after data, Bit mode
7	Reset before and after data, Bit mode

The correct value for Mode depends on the 1-wire device and the portion of the communication that is being dealt with. Consult the data sheet for the device in question to determine the correct value for Mode. In many cases, however, when using the **Oread** command, Mode should be set for either No Reset (to receive data from a transaction already started by an **Owrite**

command) or a Reset after data (to terminate the session after data is received). However, this may vary due to device and application requirements.

When using the Bit (rather than Byte) mode of data transfer, all variables in the InputData argument will only receive one bit. For example, the following code could be used to receive two bits using this mode: -

```
Dim BitVar1 as Bit
Dim BitVar2 as Bit
Oread PORTA.0, 6, [BitVar1, BitVar2]
```

In the example code shown, a value of 6 was chosen for Mode. This sets Bit transfer and Reset after data mode.

We could also have chosen to make the BitVar1 and BitVar2 variables each a Byte type, however, they would still only have received one bit each in the **Oread** command, due to the Mode that was chosen.

The compiler also has a modifier for handling a string of data, named **Str**.

The **Str** modifier is used for receiving data and placing it directly into a byte array variable.

A string is a set of bytes that are arranged or accessed in a certain order. The values 1, 2, 3 would be stored in a string with the value 1 first, followed by 2 then followed by the value 3. A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string 1 2 3 would be stored in a byte array containing three bytes (elements).

Below is an example that receives ten bytes through a 1-wire interface and stores them in the 10-byte array, MyArray: -

```
Dim MyArray[10] as Byte      ' Create a 10-byte array.
Oread DQ, 1, [Str MyArray]
Print Dec Str MyArray      ' Display the values.
```

If the amount of received characters is not enough to fill the entire array, then a formatter may be placed after the array's name, which will only receive characters until the specified length is reached. For example: -

```
Dim MyArray[10] as Byte      ' Create a 10-byte array.
Oread DQ, 1, [Str MyArray\5] ' Fill the first 5-bytes of array with data.
Print Str MyArray \5        ' Display the 5-value string.
```

The example above illustrates how to fill only the first n bytes of an array, and then how to display only the first n bytes of the array. n refers to the value placed after the backslash.

Dallas 1-Wire Protocol.

The 1-wire protocol has a well defined standard for transaction sequences. Every transaction sequence consists of four parts: -

- Initialisation.
- ROM Function Command.
- Memory Function Command.
- Transaction / Data.

Additionally, the ROM Function Command and Memory Function Command are always 8 bits wide and are sent least-significant-bit first (LSB).

The Initialisation consists of a reset pulse (generated by the master) that is followed by a presence pulse (generated by all slave devices).

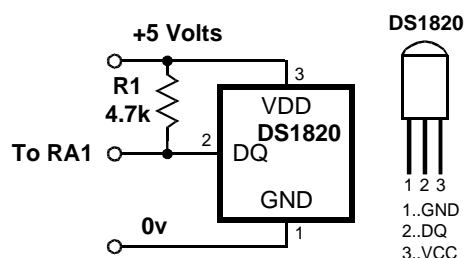
The reset pulse is controlled by the lowest two bits of the Mode argument in the Oread command. It can be made to appear before the ROM Function Command (Mode = 1), after the Transaction / Data portion (Mode = 2), before and after the entire transaction (Mode = 3) or not at all (Mode = 0).

Command	Value	Action
Read ROM	\$33	Reads the 64-bit ID of the 1-wire device. This command can only be used if there is a single 1-wire device on the line.
Match ROM	\$55	This command, followed by a 64-bit ID, allows the PICmicro to address a specific 1-wire device.
Skip ROM	\$CC	Address a 1-wire device without its 64-bit ID. This command can only be used if there is a single 1-wire device on the line.
Search ROM	\$F0	Reads the 64-bit IDs of all the 1-wire devices on the line. A process of elimination is used to distinguish each unique device.

Following the Initialisation, comes the ROM Function Command. The ROM Function Command is used to address the desired 1-wire device. The above table shows a few common ROM Function Commands. If only a single 1 wire device is connected, the Match ROM command can be used to address it. If more than one 1-wire device is attached, the PICmicro™ will ultimately have to address them individually using the Match ROM command.

The third part, the Memory Function Command, allows the PICmicro™ to address specific memory locations, or features, of the 1-wire device. Refer to the 1-wire device's data sheet for a list of the available Memory Function Commands.

Finally, the Transaction / Data section is used to read or write data to the 1-wire device. The **Oread** command will read data at this point in the transaction. A read is accomplished by generating a brief low-pulse and sampling the line within 15us of the falling edge of the pulse. This is called a 'Read Slot'.



The following program demonstrates interfacing to a Dallas Semiconductor DS1820 1-wire digital thermometer device using the compiler's 1-wire commands, and connections as per the diagram to the right.

The code reads the Counts Remaining and Counts per Degree Centigrade registers within the DS1820 device in order to provide a more accurate temperature (down to 1/10th of a degree).

```
Device = 16F84
Declare Xtal = 4
Symbol DQ = PORTA.1           ' Place the DS1820 on bit-1 of PORTA
Dim Temp as Word              ' Holds the temperature value
Dim C as Byte                 ' Holds the counts remaining value
Dim CPerD as Byte            ' Holds the Counts per degree C value
Cls                             ' Clear the LCD before we start
Again:
Owrite DQ, 1, [$CC, $44]      ' Send Calculate Temperature command
Repeat
    DelayMs 25                 ' Wait until conversion is complete
    Oread DQ, 4, [C]           ' Keep reading low pulses until
    Until C <> 0                ' the DS1820 is finished.
    Owrite DQ, 1, [$CC, $BE]   ' Send Read ScratchPad command
    Oread DQ, 2,[Temp.LowByte,Temp.HighByte, C, C, C, C, C, CPerD]
' Calculate the temperature in degrees Centigrade
Temp = (((Temp >> 1) * 100) - 25) + (((CPerD - C) * 100) / CPerD)
Print At 1,1, Dec Temp / 100, ".", Dec2 Temp, " ", At 1,8,"C"
Goto Again
```

Note.

The equation used in the program above will not work correctly with negative temperatures. Also note that the 4.7kΩ pull-up resistor (R1) is required for correct operation.

Inline Oread Command.

The standard structure of the **Oread** command is: -

```
Oread Pin, Mode, [ Inputdata ]
```

However, this did not allow it to be used in conditions such as **If-Then**, **While-Wend** etc. Therefore, there is now an additional structure to the **Oread** command: -

```
Var = Oread Pin, Mode
```

Operands Pin and Mode have not changed their function, but the result from the 1-wire read is now placed directly into the assignment variable.

Oread - Owrite Presence Detection.

Another important feature to both the **Oread** and **Owrite** commands is the ability to jump to a section of the program if a presence is not detected on the 1-wire bus.

Owrite Pin, Mode, Label, [Outputdata]

Oread Pin, Mode, Label, [Inputdata]

Var = **Oread** Pin, Mode, Label

The Label parameter is an optional condition, but if used, it must reference a valid BASIC label.

```
' Skip ROM search & do temp conversion
Owrite DQ, 1, NoPresence, [$CC, $44]
While Oread DQ, 4, NoPresence != 0 : Wend ' Read busy-bit,' Still busy..?
' Skip ROM search & read scratchpad memory
Owrite DQ, 1, NoPresence, [$CC, $BE]
Oread DQ, 2, NoPresence, [Temp.Lowbyte, Temp.Highbyte] ' Read two bytes
Return

NoPresence:
Print "No Presence"
Stop
```

See also : **Owrite.**

Owrite

Syntax

Owrite *Pin, Mode, [Outputdata]*

Overview

Send data to a device using the Dallas Semiconductor 1-wire protocol. The 1-wire protocol is a form of asynchronous serial communication developed by Dallas Semiconductor. It requires only one I/O pin which may be shared between multiple 1-wire devices.

Operators

Pin is a Port-Bit combination that specifies which I/O pin to use. 1-wire devices require only one I/O pin (normally called DQ) to communicate. This I/O pin will be toggled between output and input mode during the Owrite command and will be set to input mode by the end of the Owrite command.

Mode is a numeric constant (0 - 7) indicating the mode of data transfer. The Mode operator controls the placement of reset pulses and detection of presence pulses, as well as byte or bit input. See notes below.

Outputdata is a list of variables or arrays transmit individual or repeating bytes.

Example

```
Symbol DQ = PORTA.0
Owrite DQ, 1, [$4E]
```

The above example will transmit a 'reset' pulse to a 1-wire device (connected to bit 0 of PORTA) and will then detect the device's 'presence' pulse and transmit one byte (the value \$4E).

Notes

The Mode operator is used to control placement of reset pulses (and detection of presence pulses) and to designate byte or bit input. The table below shows the meaning of each of the 8 possible value combinations for Mode.

Mode Value	Effect
0	No Reset, Byte mode
1	Reset before data, Byte mode
2	Reset after data, Byte mode
3	Reset before and after data, Byte mode
4	No Reset, Bit mode
5	Reset before data, Bit mode
6	Reset after data, Bit mode
7	Reset before and after data, Bit mode

The correct value for Mode depends on the 1-wire device and the portion of the communication you're dealing with. Consult the data sheet for the device in question to determine the correct value for Mode. In many cases, however, when using the **Owrite** command, Mode should be set for a Reset before data (to initialise the transaction). However, this may vary due to device and application requirements.

When using the Bit (rather than Byte) mode of data transfer, all variables in the InputData argument will only receive one bit. For example, the following code could be used to receive two bits using this mode: -

```
Dim BitVar1 as Bit
Dim BitVar2 as Bit
Owrite PORTA.0, 6, [BitVar1, BitVar2]
```

In the example code shown, a value of 6 was chosen for Mode. This sets Bit transfer and Reset after data mode. We could also have chosen to make the BitVar1 and BitVar2 variables each a Byte type, however, they would still only use their lowest bit (Bit0) as the value to transmit in the **Owrite** command, due to the Mode value chosen.

The Str Modifier

The **Str** modifier is used for transmitting a string of bytes from a byte array variable. A string is a set of bytes sized values that are arranged or accessed in a certain order. The values 1, 2, 3 would be stored in a string with the value 1 first, followed by 2 then followed by the value 3. A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string 1,2,3 would be stored in a byte array containing three bytes (elements).

Below is an example that sends four bytes (from a byte array) through bit-0 of PORTA: -

```
Dim MyArray[10] as Byte      ' Create a 10-byte array.
MyArray [0] = $CC           ' Load the first 4 bytes of the array
MyArray [1] = $44           ' With the data to send
MyArray [2] = $CC
MyArray [3] = $4E
Owrite PORTA.0, 1, [Str MyArray\4] ' Send 4-byte string.
```

Note that we use the optional \n argument of **Str**. If we didn't specify this, the PICmicro™ would try to keep sending characters until all 10 bytes of the array were transmitted. Since we do not wish all 10 bytes to be transmitted, we chose to tell it explicitly to only send the first 4 bytes.

The above example may also be written as: -

```
Dim MyArray [10] as Byte      ' Create a 10-byte array.
Str MyArray = $CC,$44,$CC,$4E ' Load the first 4 bytes of the array
Owrite PORTA.0, 1, [Str MyArray\4] ' Send 4-byte string.
```

The above example, has exactly the same function as the previous one. The only difference is that the string is now constructed using the **Str** as a command instead of a modifier.

See also : [Oread for example code, and 1-wire protocol.](#)

Pixel

Syntax

Variable = **Pixel** *Ypos*, *Xpos*

Overview

Read the condition of an individual pixel from a graphic LCD. The returned value will be 1 if the pixel is set, and 0 if the pixel is clear.

Operators

Variable is a user defined variable.

Xpos can be a constant, variable, or expression, pointing to the X-axis location of the pixel to examine. This must be a value of 0 to the X resolution of the LCD. Where 0 is the far left row of pixels.

Ypos can be a constant, variable, or expression, pointing to the Y-axis location of the pixel to examine. This must be a value of 0 to the Y resolution of the LCD. Where 0 is the top column of pixels.

Example

```
' Read a line of pixels from a Samsung KS0108 graphic LCD
Device = 16F877
Declare LCD_Type = Graphic      ' Use a Graphic LCD
Declare Internal_Font = Off     ' Use an external chr set
Declare Font_Addr = 0          ' Eeprom's address is 0

' Graphic LCD Pin Assignments
Declare LCD_DTPort = PORTD
Declare LCD_RSPin = PORTC.2
Declare LCD_RWPin = PORTE.0
Declare LCD_ENPin = PORTC.5
Declare LCD_CS1Pin = PORTE.1
Declare LCD_CS2Pin = PORTE.2

' Character set eeprom Pin Assignments
Declare SDA_Pin = PORTC.4
Declare SCL_Pin = PORTC.3

Dim Xpos as Byte
Dim Ypos as Byte
Dim Result as Byte

ADCON1 = 7      ' PORTA and PORTE to all digital mode
Cls
Print At 0, 0, "Testing 1-2-3"
,
' Read the top row and display the result
,
For Xpos = 0 to 127
Result = Pixel 0, Xpos      ' Read the top row
Print At 1, 0, Dec Result
DelayMs 400
Next
Stop
```

See also : LCDread, LCDwrite, Plot, UnPlot. See Print for circuit.

Plot

Syntax

Plot *Ypos*, *Xpos*

Overview

Set an individual pixel on a graphic LCD.

Operators

Xpos can be a constant, variable, or expression, pointing to the X-axis location of the pixel to set. This must be a value of 0 to the X resolution of the LCD. Where 0 is the far left row of pixels.

Ypos can be a constant, variable, or expression, pointing to the Y-axis location of the pixel to set. This must be a value of 0 to the Y resolution of the LCD. Where 0 is the top column of pixels.

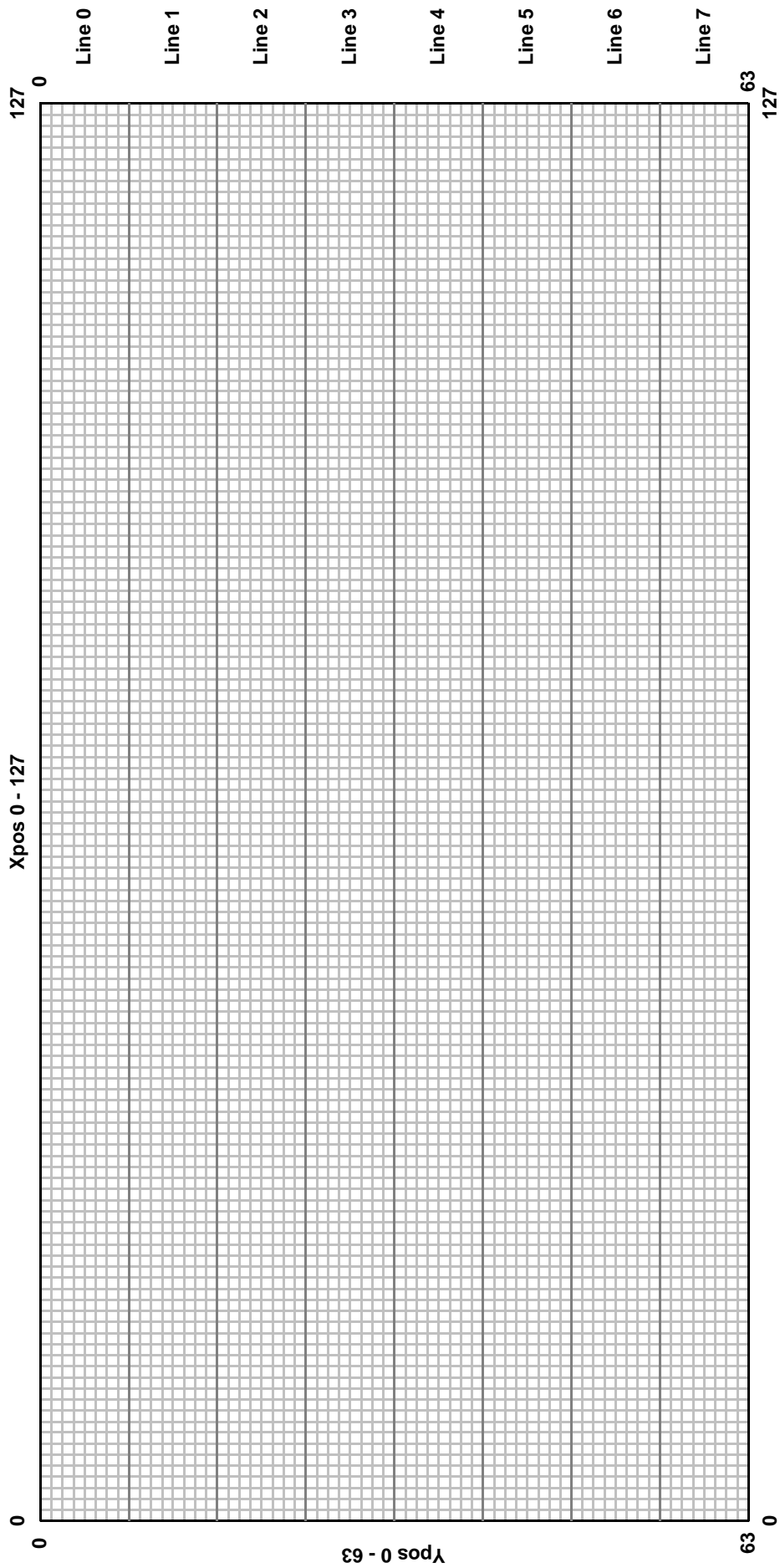
Example

```
Device = 16F877
Declare LCD_Type = Graphic      ' Use a Samsung Graphic LCD
,
' Graphic LCD Pin Assignments
,
Declare LCD_DTPort = PORTD
Declare LCD_RSPin = PORTC.2
Declare LCD_RWPin = PORTE.0
Declare LCD_ENPin = PORTC.5
Declare LCD_CS1Pin = PORTE.1
Declare LCD_CS2Pin = PORTE.2

Dim Xpos as Byte
ADCON1 = 7                      ' PORTA and PORTE to all digital mode
,
' Draw a line across the LCD
,
While                            ' Create an infinite loop
For Xpos = 0 to 127
    Plot 20, Xpos
    DelayMs 10
Next
,
' Now erase the line
,
For Xpos = 0 to 127
    UnPlot 20, Xpos
    DelayMs 10
Next
Wend
```

See also : LCDread, LCDwrite, Pixel, UnPlot. See Print for circuit.

Graphic LCD pixel configuration for a 128x64 resolution display.



Pop

Syntax

Pop *Variable*, {*Variable*, *Variable* etc}

Overview

Pull a single variable or multiple variables from a software stack.

If the **Pop** command is issued without a following variable, it will implement the assembler mnemonic **Pop**, which manipulates the PICmicro's call stack.

Operators

Variable is a user defined variable of type **Bit**, **Byte**, **Word**, **Dword**, **Float**, **Array**, or **String**.

The amount of bytes pushed on to the stack varies with the variable type used. The list below shows how many bytes are pushed for a particular variable type, and their order.

Bit	1 Byte is popped containing the value of the bit pushed.
Byte	1 Byte is popped containing the value of the byte pushed.
Byte Array	1 Byte is popped containing the value of the byte pushed.
Word	2 Bytes are popped. Low Byte then High Byte containing the value of the word pushed.
Word Array	2 Bytes are popped. Low Byte then High Byte containing the value of the word pushed.
Dword Array	4 Bytes are popped. Low Byte, Mid1 Byte, Mid2 Byte then High Byte containing the value of the dword pushed.
Dword	4 Bytes are popped. Low Byte, Mid1 Byte, Mid2 Byte then High Byte containing the value of the dword pushed.
Float	4 Bytes are popped. Low Byte, Mid1 Byte, Mid2 Byte then High Byte containing the value of the float pushed.
String	2 Bytes are popped. Low Byte then High Byte that point to the start address of the string previously pushed.

Example 1

' Push two variables on to the stack then retrieve them

```
Device = 18F452      ' Stack only suitable for 18F devices
Declare Stack_Size = 20 ' Create a small stack capable of holding 20 bytes

Dim Wrd as Word      ' Create a Word variable
Dim Dwd as Dword     ' Create a Dword variable

Wrd = 1234           ' Load the Word variable with a value
Dwd = 567890        ' Load the Dword variable with a value
Push Wrd, Dwd       ' Push the Word variable then the Dword variable

Clear Wrd           ' Clear the Word variable
Clear Dwd           ' Clear the Dword variable

Pop Dwd, Wrd        ' Pop the Dword variable then the Word variable
Print Dec Wrd, " ", Dec Dwd ' Display the variables as decimal
Stop
```

Example 2

```
' Push a String on to the stack then retrieve it

Device = 18F452      ' Stack only suitable for 18F devices
Declare Stack_Size = 10 ' Create a small stack capable of holding 10 bytes

Dim SourceString as String * 20 ' Create a String variable
Dim DestString as String * 20    ' Create another String variable

SourceString = "HELLO WORLD"    ' Load the String variable with characters

Push SourceString              ' Push the String variable's address

Pop DestString                 ' Pop the previously pushed String into DestString
Print DestString               ' Display the string, which will be "HELLO WORLD"
Stop
```

Example 3

```
' Push a Quoted character string on to the stack then retrieve it

Device = 18F452      ' Stack only suitable for 18F devices
Declare Stack_Size = 10 ' Create a small stack capable of holding 10 bytes

Dim DestString as String * 20 ' Create a String variable

Push "HELLO WORLD"        ' Push the Quoted String of Characters on to the stack

Pop DestString            ' Pop the previously pushed String into DestString
Print DestString          ' Display the string, which will be "HELLO WORLD"
Stop
```

See also : **Push, Gosub, Return, See Push** for technical details of stack manipulation.

Pot

Syntax

`Variable = Pot Pin, Scale`

Overview

Read a potentiometer, thermistor, photocell, or other variable resistance.

Operators

Variable is a user defined variable.

Pin is a Port.Pin constant that specifies the I/O pin to use.

Scale is a constant, variable, or expression, used to scale the instruction's internal 16-bit result. The 16-bit reading is multiplied by (scale/ 256), so a *scale* value of 128 would reduce the range by approximately 50%, a scale of 64 would reduce to 25%, and so on.

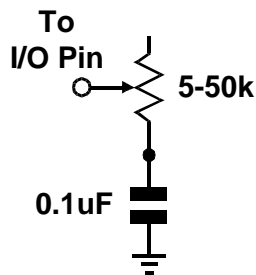
Example

```
Dim Var1 as Byte
Loop:
  Var1 = Pot PORTB.0, 100 ' Read potentiometer on pin 0 of PORTB.
  Print Dec Var1, " " ' Display the potentiometer reading
  Goto Loop ' Repeat the process.
```

Notes

Internally, the **Pot** instruction calculates a 16-bit value, which is scaled down to an 8-bit value. The amount by which the internal value must be scaled varies with the size of the resistor being used.

The pin specified by **Pot** must be connected to one side of a resistor, whose other side is connected through a capacitor to ground. A resistance measurement is taken by timing how long it takes to discharge the capacitor through the resistor.



The value of *scale* must be determined by experimentation, however, this is easily accomplished as follows: -

Set the device under measure, the pot in this instance, to maximum resistance and read it with *scale* set to 255. The value returned in Var1 can now be used as *scale*: -

```
Var1 = Pot PORTB.0, 255
```

See also : Adin, RCin.

Print

Syntax

Print *Item* {, *Item...* }

Overview

Send Text to an LCD module using the Hitachi 44780 controller or a graphic LCD based on the Samsung KS0108, or Toshiba T6963 chipsets.

Operators

Item may be a constant, variable, expression, modifier, or string list.

There are no operators as such, instead there are *modifiers*. For example, if an at sign '@' precedes an *Item*, the ASCII representation for each digit is sent to the LCD.

The modifiers are listed below: -

Modifier	Operation
----------	-----------

At ypos (1 to n),xpos(1 to n)	Position the cursor on the LCD
--------------------------------------	--------------------------------

Cl	Clear the LCD (also creates a 30ms delay)
-----------	---

Bin{1..32}	Display binary digits
-------------------	-----------------------

Dec{1..10}	Display decimal digits
-------------------	------------------------

Hex{1..8}	Display hexadecimal digits
------------------	----------------------------

Sbin{1..32}	Display signed binary digits
--------------------	------------------------------

Sdec{1..10}	Display signed decimal digits
--------------------	-------------------------------

Shex{1..8}	Display signed hexadecimal digits
-------------------	-----------------------------------

lbin{1..32}	Display binary digits with a preceding '%' identifier
--------------------	---

ldec{1..10}	Display decimal digits with a preceding '#' identifier
--------------------	--

lhex{1..8}	Display hexadecimal digits with a preceding '\$' identifier
-------------------	---

lSbin{1..32}	Display signed binary digits with a preceding '%' identifier
---------------------	--

lSdec{1..10}	Display signed decimal digits with a preceding '#' identifier
---------------------	---

lShex{1..8}	Display signed hexadecimal digits with a preceding '\$' identifier
--------------------	--

Rep c\n	Display character c repeated n times
----------------	--------------------------------------

Str array\n	Display all or part of an array
--------------------	---------------------------------

Cstr cdata	Display string data defined in a Cdata statement.
-------------------	---

The numbers after the **Bin**, **Dec**, and **Hex** modifiers are optional. If they are omitted, then the default is all the digits that make up the value will be displayed.

If a floating point variable is to be displayed, then the digits after the **Dec** modifier determine how many remainder digits are printed. i.e. numbers after the decimal point.

```
Dim Flt as Float
Flt = 3.145
Print Dec2 Flt    ' Display 2 values after the decimal point
```

The above program will display 3.14

If the digit after the **Dec** modifier is omitted, then 3 values will be displayed after the decimal point.

```
Dim Flt as Float
Flt = 3.1456
Print Dec Flt      ' Display 3 values after the decimal point
```

The above program will display 3.145

There is no need to use the **Sdec** modifier for signed floating point values, as the compiler's **Dec** modifier will automatically display a minus result: -

```
Dim Flt as Float
Flt = -3.1456
Print Dec Flt      ' Display 3 values after the decimal point
```

The above program will display -3.145

Hex or **Bin** modifiers cannot be used with floating point values or variables.

The Xpos and Ypos values in the **At** modifier both start at 1. For example, to place the text "HELLO WORLD" on line 1, position 1, the code would be: -

```
Print At 1, 1, "HELLO WORLD"
```

Example 1

```
Dim Var1 as Byte
Dim Wrđ as Word
Dim Dwd as Dword

Print "Hello World"      ' Display the text "Hello World"
Print "Var1= ", Dec Var1  ' Display the decimal value of Var1
Print "Var1= ", Hex Var1  ' Display the hexadecimal value of Var1
Print "Var1= ", Bin Var1  ' Display the binary value of Var1
Print "Dwd= ", Hex6 Dwd   ' Display 6 hex characters of a Dword variable
```

Example 2

```
' Display a negative value on the LCD.
Symbol Negative = -200
Print At 1, 1, Sdec Negative
```

Example 3

```
' Display a negative value on the LCD with a preceding identifier.
Print At 1, 1, IShex -$1234
```

Example 3 will produce the text "\$-1234" on the LCD.

Some PICmicros such as the 16F87x, and 18FXXX range have the ability to read and write to their own flash memory. And although writing to this memory too many times is unhealthy for the PICmicro™, reading this memory is both fast, and harmless. Which offers a unique form of data storage and retrieval, the **Cdata** command proves this, as it uses the mechanism of reading and storing in the PICmicro's flash memory.

Combining the unique features of the 'self modifying PICmicro's' with a string format, the compiler is capable of reducing the overhead of printing, or transmitting large amounts of text data. The **Cstr** modifier may be used in commands that deal with text processing i.e. **Serout**, **Hrsout**, and **RSOUT** etc.

The **Cstr** modifier is used in conjunction with the **Cdata** command. The **Cdata** command is used for initially creating the string of characters: -

```
String1: Cdata "HELLO WORLD", 0
```

The above line of code will create, in flash memory, the values that make up the ASCII text "HELLO WORLD", at address String1. Note the null terminator after the ASCII text.

null terminated means that a zero (null) is placed at the end of the string of ASCII characters to signal that the string has finished.

To display this string of characters, the following command structure could be used: -

```
Print Cstr String1
```

The label that declared the address where the list of Cdata values resided, now becomes the string's name. In a large program with lots of text formatting, this type of structure can save quite literally hundreds of bytes of valuable code space.

Try both these small programs, and you'll see that using **Cstr** saves a few bytes of code: -

First the standard way of displaying text: -

```
Device = 16F877
Cls
Print "HELLO WORLD"
Print "HOW ARE YOU?"
Print "I AM FINE!"
Stop
```

Now using the **Cstr** modifier: -

```
Cls
Print Cstr TEXT1
Print Cstr TEXT2
Print Cstr TEXT3
Stop
```

```
TEXT1: Cdata "HELLO WORLD", 0
TEXT2: Cdata "HOW ARE YOU?", 0
TEXT3: Cdata "I AM FINE!", 0
```

Again, note the null terminators after the ASCII text in the **Cdata** commands. Without these, the PICmicro™ will continue to transmit data in an endless loop.

The term 'virtual string' relates to the fact that a string formed from the Cdata command cannot be written to, but only read from.

The **Str** modifier is used for sending a string of bytes from a byte array variable. A string is a set of bytes sized values that are arranged or accessed in a certain order. The values 1, 2, 3 would be stored in a string with the value 1 first, followed by 2 then followed by the value 3. A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string 1,2,3 would be stored in a byte array containing three bytes (elements).

Below is an example that displays four bytes (from a byte array): -

```
Dim MyArray[10] as Byte      ' Create a 10-byte array.
MyArray [0] = "H"           ' Load the first 5 bytes of the array
MyArray [1] = "E"           ' With the data to send
MyArray [2] = "L"
MyArray [3] = "L"
MyArray [4] = "O"
Print Str MyArray\5         ' Display a 5-byte string.
```

Note that we use the optional `\n` argument of **Str**. If we didn't specify this, the PICmicro™ would try to keep sending characters until all 10 bytes of the array were transmitted. Since we do not wish all 10 bytes to be transmitted, we chose to tell it explicitly to only send the first 5 bytes.

The above example may also be written as: -

```
Dim MyArray[10] as Byte      ' Create a 10-byte array.
Str MyArray = "HELLO"        ' Load the first 5 bytes of the array
Print Str MyArray\5         ' Send 5-byte string.
```

The above example, has exactly the same function as the previous one. The only difference is that the string is now constructed using **Str** as a command instead of a modifier.

Declares

There are several Declares for use with an alphanumeric LCD and **Print**: -

Declare LCD_Type 0 or 1 or 2, Alpha or Graphic or Samsung or Toshiba

Inform the compiler as to the type of LCD that the **Print** command will output to. If **Graphic**, **Samsung** or 1 is chosen then any output by the **Print** command will be directed to a graphic LCD based on the Samsung KS0108 chipset. A value of 2, or the text **Toshiba**, will direct the output to a graphic LCD based on the Toshiba T6963 chipset. A value of 0 or **Alpha**, or if the **Declare** is not issued, will target the standard Hitachi alphanumeric LCD type

Targeting the graphic LCD will also enable commands such as **Plot**, **UnPlot**, **LCDread**, **LCDwrite**, **Pixel**, **Box**, **Circle** and **Line**.

Declare LCD_DTPin Port . Pin

Assigns the Port and Pins that the LCD's DT (data) lines will attach to.

The LCD may be connected to the PICmicro™ using either a 4-bit bus or an 8-bit bus. If an 8-bit bus is used, all 8 bits must be on one port. If a 4-bit bus is used, it must be connected to either the bottom 4 or top 4 bits of one port. For example: -

```
Declare LCD_DTPin PORTB.4    ' Used for 4-line interface.

Declare LCD_DTPin PORTB.0    ' Used for 8-line interface.
```

In the previous examples, PORTB is only a personal preference. The LCD's DT lines may be attached to any valid port on the PICmicro™. If the **Declare** is not used in the program, then the default Port and Pin is PORTB.4, which assumes a 4-line interface.

Declare LCD_ENPin Port . Pin

Assigns the Port and Pin that the LCD's EN line will attach to.

If the **Declare** is not used in the program, then the default Port and Pin is PORTB.2.

Declare LCD_RSPin Port . Pin

Assigns the Port and Pins that the LCD's RS line will attach to.

If the **Declare** is not used in the program, then the default Port and Pin is PORTB.3.

Declare LCD_Interface 4 or 8

Inform the compiler as to whether a 4-line or 8-line interface is required by the LCD.

If the **Declare** is not used in the program, then the default interface is a 4-line type.

Declare LCD_Lines 1, 2, or 4

Inform the compiler as to how many lines the LCD has.

LCD's come in a range of sizes, the most popular being the 2 line by 16 character types. However, there are 4 line types as well. Simply place the number of lines that the particular LCD has, into the declare.

If the **Declare** is not used in the program, then the default number of lines is 2.

Notes

If no modifier precedes an item in a **Print** command, then the character's value is sent to the LCD. This is useful for sending control codes to the LCD. For example: -

```
Print $FE, 128
```

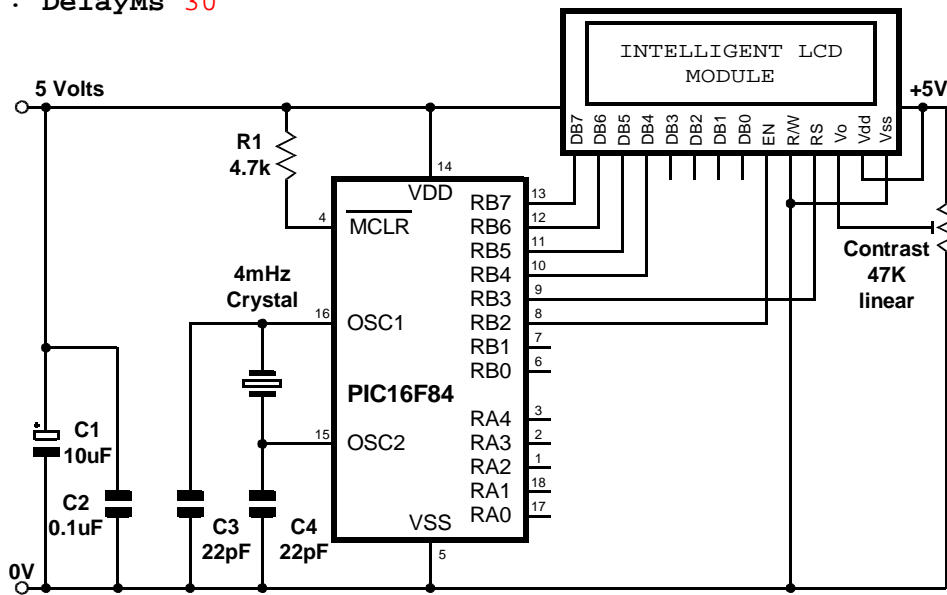
Will move the cursor to line 1, position 1 (HOME).

Below is a list of some useful control commands: -

Control Command	Operation
\$FE, 1	Clear display
\$FE, 2	Return home (beginning of first line)
\$FE, \$0C	Cursor off
\$FE, \$0E	Underline cursor on
\$FE, \$0F	Blinking cursor on
\$FE, \$10	Move cursor left one position
\$FE, \$14	Move cursor right one position
\$FE, \$C0	Move cursor to beginning of second line
\$FE, \$94	Move cursor to beginning of third line (if applicable)
\$FE, \$D4	Move cursor to beginning of fourth line (if applicable)

Note that if the command for clearing the LCD is used, then a small delay should follow it: -

```
Print $FE, 1 : DelayMs 30
```



The above diagram shows the default connections for an alphanumeric LCD module. In this instance, connected to the 16F84 PICmicro™.

Using a Samsung KS0108 Graphic LCD

Once a Samsung graphic LCD has been chosen using the **Declare LCD_Type** directive, all **Print** outputs will be directed to that LCD.

The standard modifiers used by an alphanumeric LCD may also be used with the graphics LCD. Most of the above modifiers still work in the expected manner, however, the **At** modifier now starts at Ypos 0 and Xpos 0, where values 0,0 will be the top left corner of the LCD.

There are also four new modifiers. These are: -

- Font** 0 to n Choose the nth font, if available
- Inverse** 0-1 Invert the characters sent to the LCD
- or 0-1 Or the new character with the original
- Xor** 0-1 Xor the new character with the original

Once one of the four new modifiers has been enabled, all future **Print** commands will use that particular feature until the modifier is disabled. For example: -

```
' Enable inverted characters from this point
Print At 0, 0, Inverse 1, "Hello World"
Print At 1, 0, "Still Inverted"
' Now use normal characters
Print At 2, 0, Inverse 0, "Normal Characters"
```

If no modifiers are present, then the character's ASCII representation will be displayed: -

```
' Print characters A and B
Print At 0, 0, 65, 66
```

Declares

There are nine declares associated with a Samsung graphic LCD.

Declare LCD_DTPort Port

Assign the port that will output the 8-bit data to the graphic LCD.

If the **Declare** is not used, then the default port is PORTD.

Declare LCD_RWPin Port . Pin

Assigns the Port and Pin that the graphic LCD's RW line will attach to.

If the **Declare** is not used in the program, then the default Port and Pin is PORTE.0.

Declare LCD_CS1Pin Port . Pin

Assigns the Port and Pin that the graphic LCD's CS1 line will attach to.

If the **Declare** is not used in the program, then the default Port and Pin is PORTC.0.

Declare LCD_CS2Pin Port . Pin

Assigns the Port and Pin that the graphic LCD's CS2 line will attach to.

If the **Declare** is not used in the program, then the default Port and Pin is PORTC.2.

Note

Along with the new declares, two of the existing LCD declares must also be used. Namely, RS_Pin and EN_Pin.

Declare Internal_Font On - Off, 1 or 0

The graphic LCDs that are compatible with Proton are non-intelligent types, therefore, a separate character set is required. This may be in one of two places, either externally, in an I²C eeprom, or internally in a **Cdata** table.

If the **Declare** is omitted from the program, then an external font is the default setting.

If an external font is chosen, the I²C eeprom must be connected to the specified SDA and SCL pins (as dictated by **Declare** SDA and **Declare** SCL).

If an internal font is chosen, it must be on a PICmicro™ device that has self modifying code features, such as the 16F87X range.

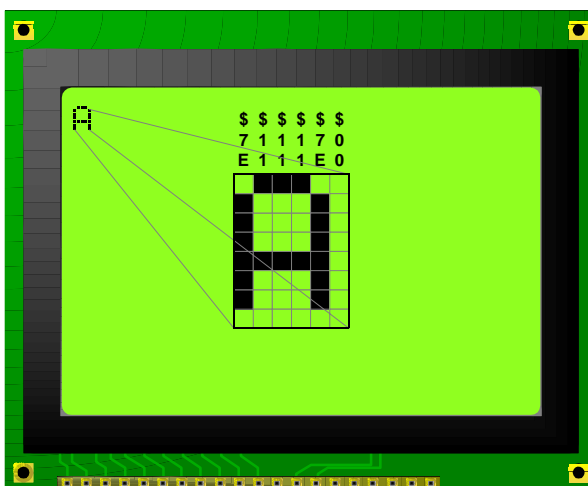
The **Cdata** table that contains the font must have a label, named Font_Table: preceding it. For example: -

```
Font_Table:- '{ data for characters 0 to 64 }
    Cdata $7E, $11, $11, $11, $7E, $00 ' Chr 65 "A"
    Cdata $7F, $49, $49, $49, $36, $00 ' Chr 66 "B"
    { rest of font table }
```

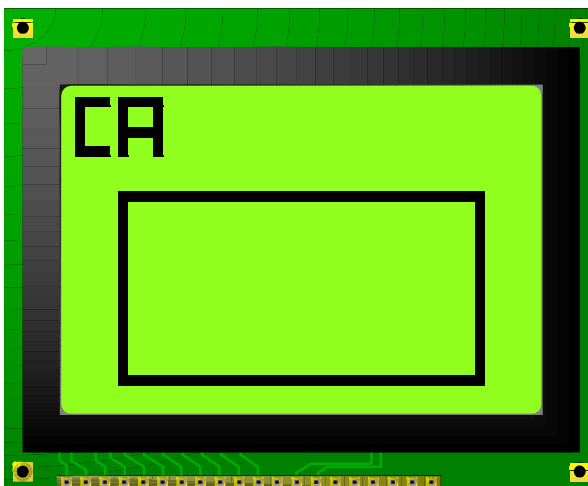
Notice the dash after the font's label, this disables any bank switching code that may otherwise disturb the location in memory of the **Cdata** table.

The font table may be anywhere in memory, however, it is best placed after the main program code.

The font is built up of an 8x6 cell, with only 5 of the 6 rows, and 7 of the 8 columns being used for alphanumeric characters. See the diagram below.



If a graphic character is chosen (chr 0 to 31), the whole of the 8x6 cell is used. In this way, large fonts and graphics may be easily constructed.



The character set itself is 128 characters long (0 -127). Which means that all the ASCII characters are present, including \$, %, &, # etc.

There are two programs on the compiler's CDROM, that are for use with internal and external fonts. **Int_Font.bas**, contains a **Cdata** table that may be cut and pasted into your own program if an internal font is chosen. **Ext_Font.bas**, writes the character set to a 24LC32 I²C eeprom for use with an external font. Both programs are fully commented.

Declare Font_Addr 0 to 7

Set the slave address for the I²C eeprom that contains the font.

When an external source for the font is used, it may be on any one of 8 eeproms attached to the I²C bus. So as not to interfere with any other eeproms attached, the slave address of the eeprom carrying the font code may be chosen.

If the **Declare** is omitted from the program, then address 0 is the default slave address of the font eeprom.

Declare GLCD_CS_Invert On - Off, 1 or 0

Some graphic LCD types have inverters on the CS lines. Which means that the LCD displays left-hand data on the right side, and vice-versa. The **GLCD_CS_Invert Declare**, adjusts the library LCD handling subroutines to take this into account.

Declare GLCD_Strobe_Delay 0 to 65535 microseconds (us).

If a noisy circuit layout is unavoidable when using a graphic LCD, then the above **Declare** may be used. This will create a delay between the Enable line being strobed. This can ease random data being produced on the LCD's screen. See below for more details on circuit layout for graphic LCDs.

If the **Declare** is not used in the program, then no delay is created between strobes, and the LCD is accessed at full efficiency.

Declare GLCD_Read_Delay 0 to 65535 microseconds (us).

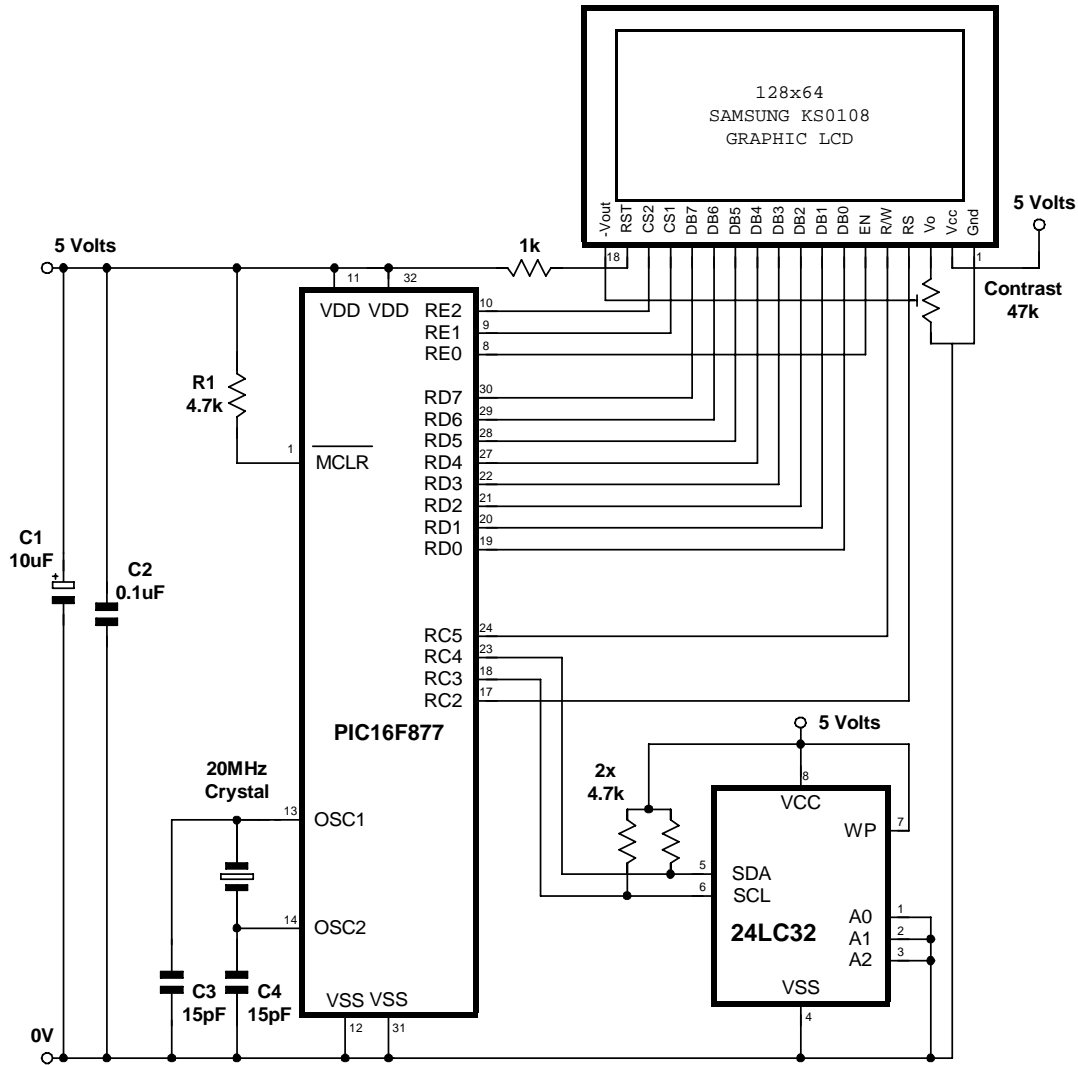
Create a delay of n microseconds between strobing the EN line of the graphic LCD, when reading from the GLCD. This can help noisy, or badly decoupled circuits overcome random bits being examined. The default if the **Declare** is not used in the BASIC program is a delay of 0.

Important

Because of the complexity involved with interfacing to the Samsung graphic LCD, **six** of the eight stack levels available on the 14-bit core devices, are used when the **Print** command is issued with an external font. Therefore, be aware that if **Print** is used within a subroutine, you must limit the amount of subroutine nesting that may take place.

If an internal font is implemented on a Samsung graphic LCD, then only **four** stack levels are used.

If any of the LCD's pins are attached to any of the microcontroller's analogue pins. i.e. PORTA or PORTE, then these pins must be set to digital by manipulating the appropriate SFRs (Special Function Registers)



The diagram above shows a typical circuit arrangement for an external font with a Samsung KS0108 graphic LCD. The eeprom has a slave address of 0. If an internal font is used, then the eeprom may be omitted.

Using a Toshiba T6963 Graphic LCD

Once a Toshiba graphic LCD has been chosen using the **Declare LCD_Type** directive, all **Print** outputs will be directed to that LCD.

The standard modifiers used by an alphanumeric LCD may also be used with the graphics LCD. Most of the modifiers still work in the expected manner, however, the **At** modifier now starts at Ypos 0 and Xpos 0, where values 0,0 correspond to the top left corner of the LCD.

The Samsung modifiers **Font**, **Inverse**, **Or**, and **Xor** are not supported because of the method Toshiba LCD's using the T6963 chipset implement text and graphics.

There are several **Declares** for use with a Toshiba graphic LCD, some optional and some mandatory.

Declare LCD_DTPort Port

Assign the port that will output the 8-bit data to the graphic LCD.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_WRPin Port . Pin

Assigns the Port and Pin that the graphic LCD's WR line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_RDPin Port . Pin

Assigns the Port and Pin that the graphic LCD's RD line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_CEPin Port . Pin

Assigns the Port and Pin that the graphic LCD's CE line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_CDPin Port . Pin

Assigns the Port and Pin that the graphic LCD's CD line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_RSTPin Port . Pin

Assigns the Port and Pin that the graphic LCD's RST line will attach to.

The LCD's RST (Reset) **Declare** is optional and if omitted from the BASIC code the compiler will not manipulate it. However, if not used as part of the interface, you must set the LCD's RST pin high for normal operation.

Declare LCD_X_Res 0 to 255

LCD displays using the T6963 chipset come in varied screen sizes (resolutions). The compiler must know how many horizontal pixels the display consists of before it can build its library sub-routines.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_Y_Res 0 to 255

LCD displays using the T6963 chipset come in varied screen sizes (resolutions). The compiler must know how many vertical pixels the display consists of before it can build its library subroutines.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_Font_Width 6 or 8

The Toshiba T6963 graphic LCDs have two internal font sizes, 6 pixels wide by eight high, or 8 pixels wide by 8 high. The particular font size is chosen by the LCD's FS pin. Leaving the FS pin floating or bringing it high will choose the 6 pixel font, while pulling the FS pin low will choose the 8 pixel font. The compiler must know what size font is required so that it can calculate screen and RAM boundaries.

Note that the compiler does not control the FS pin and it is down to the circuit layout whether or not it is pulled high or low. There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_RAM_Size 1024 to 65535

Toshiba graphic LCDs contain internal RAM used for Text, Graphic or Character Generation. The amount of RAM is usually dictated by the display's resolution. The larger the display, the more RAM is normally present. Standard displays with a resolution of 128x64 typically contain 4096 bytes of RAM, while larger types such as 240x64 or 190x128 typically contain 8192 bytes or RAM. The display's datasheet will inform you of the amount of RAM present.

If this **Declare** is not issued within the BASIC program, the default setting is 8192 bytes.

Declare LCD_Text_Pages 1 to n

As mentioned above, Toshiba graphic LCDs contain RAM that is set aside for text, graphics or characters generation. In normal use, only one page of text is all that is required, however, the compiler can re-arrange its library subroutines to allow several pages of text that is continuous. The amount of pages obtainable is directly proportional to the RAM available within the LCD itself. Larger displays require more RAM per page, therefore always limit the amount of pages to only the amount actually required or unexpected results may be observed as text, graphic and character generator RAM areas merge.

This **Declare** is purely optional and is usually not required. The default is 3 text pages if this **Declare** is not issued within the BASIC program.

Declare LCD_Graphic_Pages 1 to n

Just as with text, the Toshiba graphic LCDs contain RAM that is set aside for graphics. In normal use, only one page of graphics is all that is required, however, the compiler can re-arrange its library subroutines to allow several pages of graphics that is continuous. The amount of pages obtainable is directly proportional to the RAM available within the LCD itself. Larger displays require more RAM per page, therefore always limit the amount of pages to only the amount actually required or unexpected results may be observed as text, graphic and character generator RAM areas merge.

This **Declare** is purely optional and is usually not required. The default is 1 graphics page if this **Declare** is not issued within the BASIC program.

Declare LCD_Text_Home_Address 0 to n

The RAM within a Toshiba graphic LCD is split into three distinct uses, text, graphics and character generation. Each area of RAM must not overlap or corruption will appear on the display as one uses the other's assigned space. The compiler's library subroutines calculate each area of RAM based upon where the text RAM starts. Normally the text RAM starts at address 0, however, there may be occasions when it needs to be set a little higher in RAM. The order of RAM is; Text, Graphic, then Character Generation.

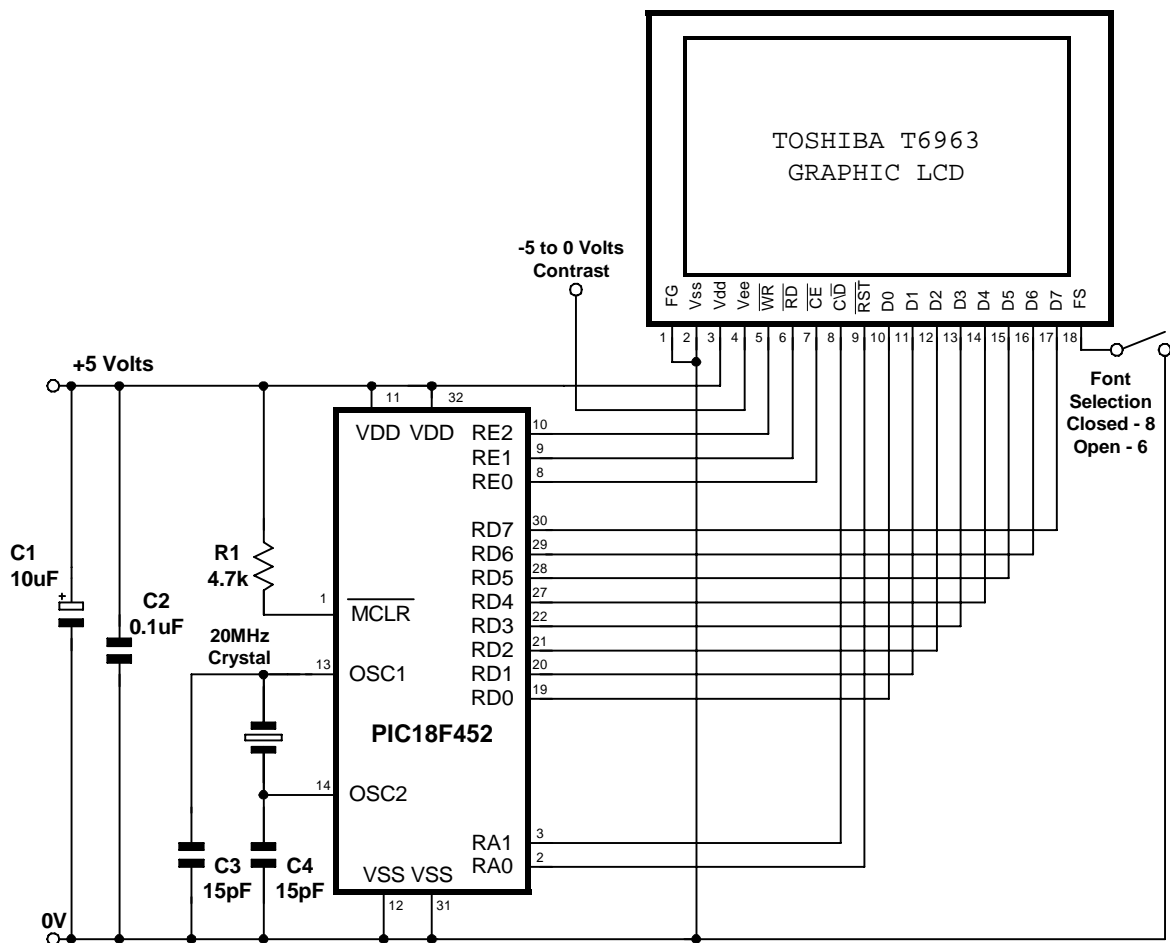
This **Declare** is purely optional and is usually not required. The default is the text RAM starting at address 0 if this **Declare** is not issued within the BASIC program.

Notes

Unlike interfacing to the Samsung graphic LCD, only **four** of the eight stack levels available on the 14-bit core devices, are used when the **Print** command is issued.

If any of the LCD's pins are attached to any of the PICmicro's analogue pins. i.e. PORTA or PORTE, then these pins must be set to digital by manipulating the appropriate SFRs (Special Function Registers).

The diagram below shows a typical circuit for an interface with a Toshiba T6963 graphic LCD.



Ptr8, Ptr16, Ptr32

Syntax

Variable = **Ptr8** (*Address*)

Variable = **Ptr16** (*Address*)

Variable = **Ptr32** (*Address*)

or

Ptr8 (*Address*) = *Variable*

Ptr16 (*Address*) = *Variable*

Ptr32 (*Address*) = *Variable*

Overview

Indirectly address RAM for loading or retrieving using a variable to hold the 16-bit address on enhanced 14-bit core devices and 18F devices.

Operands

Variable is a user defined variable that holds the result of the indirectly address RAM area, or the variable to place into the indirectly addressed RAM area.

Address is a **Word** variable that holds the 16-bit address of the RAM area of interest.

Address can also post or pre increment or decrement:

- (MyAddress++) Post increment MyAddress after retrieving it's RAM location.
- (MyAddress --) Post decrement MyAddress after retrieving it's RAM location.
- (++MyAddress) Pre increment MyAddress before retrieving it's RAM location.
- (--MyAddress) Pre decrement MyAddress before retrieving it's RAM location.

Ptr8 will load or retrieve a value with an optional 8-bit post or pre increment or decrement.

Ptr16 will load or retrieve a value with an optional 16-bit post or pre increment or decrement.

Ptr32 will load or retrieve a value with an optional 32-bit post or pre increment or decrement.

8-bit Example.

```
'  
' Load and Read 8-bit values indirectly from/to RAM  
'  
Device = 18F25K20           ' Choose an 18F device  
Declare Xtal = 16  
  
Declare Hserial_Baud = 9600   ' Set baud rate to 9600  
Declare Hserial_TXSTA = %00100000 ' Enable transmit  
  
Dim MyByteArray[20] As Byte   ' Create a byte array  
Dim MyByte As Byte           ' Create a byte variable  
Dim bIndex As Byte  
Dim wAddress as Word         ' Create a variable to hold address
```

Main:

```
'  
' Load into RAM  
'  
wAddress = AddressOf(MyByteArray) ' Load wAddress with address of array  
For bIndex = 19 To 0 Step -1 ' Create a loop  
    Ptr8(wAddress++) = bIndex ' Load RAM with address post increment  
Next  
'  
' Read from RAM  
'  
wAddress = AddressOf(MyByteArray) ' Load wAddress with address of array  
While ' Create a loop  
    MyByte = Ptr8(wAddress++) ' Retrieve from RAM with post increment  
    HRSOut Dec MyByte, 13 ' Transmit the byte read from RAM  
    If MyByte = 0 Then Break ' Exit when a null(0) is read from RAM  
Wend
```

16-bit Example.

```
' Load and Read 16-bit values indirectly from/to RAM  
'  
Device = 18F25K20 ' Choose an 18F device  
Declare Xtal = 16  
  
Declare Hserial_Baud = 9600 ' Set baud rate to 9600  
Declare Hserial_TXSTA = %00100000 ' Enable transmit  
  
Dim MyWordArray[20] As Word ' Create a word array  
Dim MyWord As Word ' Create a word variable  
Dim bIndex As Byte  
Dim wAddress as Word ' Create a variable to hold the address
```

Main:

```
'  
' Load into RAM  
'  
wAddress = AddressOf(MyWordArray) ' Load wAddress with address of array  
For bIndex = 19 To 0 Step -1 ' Create a loop  
    Ptr16(wAddress++) = bIndex ' Load RAM with address post increment  
Next  
'  
' Read from RAM  
'  
wAddress = AddressOf(MyWordArray) ' Load wAddress with address of array  
While ' Create a loop  
    MyWord = Ptr16(wAddress++) ' Retrieve from RAM with post increment  
    HRSOut Dec MyWord, 13 ' Transmit the word read from RAM  
    If MyWord = 0 Then Break ' Exit when a null(0) is read from RAM  
Wend
```

32-bit Example.

```
' Load and Read 32-bit values indirectly from RAM
',
Device = 18F25K20           ' Choose an 18F device
Declare Xtal = 16

Declare Hserial_Baud = 9600 ' Set baud rate to 9600
Declare Hserial_TXSTA = %00100000 ' Enable transmit

Dim MyDwordArray[20] As Dword ' Create a dword array
Dim MyDword As Dword          ' Create a dword variable
Dim bIndex As Byte
Dim wAddress as Word          ' Create a variable to hold the address

Main:
',
' Load into RAM
',
wAddress = AddressOf(MyDwordArray) ' Load wAddress with address of array
For bIndex = 19 To 0 Step -1       ' Create a loop
    Ptr32(wAddress++) = bIndex     ' Load RAM with address post increment
Next
',
' Read from RAM
',
wAddress = AddressOf(MyDwordArray) ' Load wAddress with address of array
While                               ' Create a loop
    MyDword = Ptr32(wAddress++)     ' Retrieve from RAM with post increment
    HRSOut Dec MyDword, 13          ' Transmit the dword read from RAM
    If MyDword = 0 Then Break      ' Exit when a null(0) is read from RAM
Wend
```

See also: [AddressOf.](#)

PulseIn

Syntax

Variable = **PulseIn** *Pin*, *State*

Overview

Change the specified pin to input and measure an input pulse.

Operators

Variable is a user defined variable. This may be a word variable with a range of 1 to 65535, or a byte variable with a range of 1 to 255.

Pin is a Port.Pin constant that specifies the I/O pin to use.

State is a constant (0 or 1) or name **High - Low** that specifies which edge must occur before beginning the measurement.

Example

```
Dim Var1 as Byte
Loop:
  Var1 = PulseIn PORTB.0, 1 ' Measure a pulse on pin 0 of PORTB.
  Print Dec Var1, " "      ' Display the reading
  Goto Loop                ' Repeat the process.
```

Notes

PulseIn acts as a fast clock that is triggered by a change in state (0 or 1) on the specified pin. When the state on the pin changes to the state specified, the clock starts counting. When the state on the pin changes again, the clock stops. If the state of the pin doesn't change (even if it is already in the state specified in the **PulseIn** instruction), the clock won't trigger. **PulseIn** waits a maximum of 0.65535 seconds for a trigger, then returns with 0 in *variable*.

The variable can be either a **Word** or a **Byte** . If the variable is a word, the value returned by **PulseIn** can range from 1 to 65535 units.

The units are dependant on the frequency of the crystal used. If a 4MHz crystal is used, then each unit is 10us, while a 20MHz crystal produces a unit length of 2us.

If the variable is a byte and the crystal is 4MHz, the value returned can range from 1 to 255 units of 10µs. Internally, **PulseIn** always uses a 16-bit timer. When your program specifies a byte, **PulseIn** stores the lower 8 bits of the internal counter into it. Pulse widths longer than 2550µs will give false, low readings with a byte variable. For example, a 2560µs pulse returns a reading of 256 with a word variable and 0 with a byte variable.

See also : Counter, PulseOut, RCin.

PulseOut

Syntax

PulseOut *Pin*, *Period*, { *Initial State* }

Overview

Generate a pulse on *Pin* of specified *Period*. The pulse is generated by toggling the pin twice, thus the initial state of the pin determines the polarity of the pulse. Or alternatively, the initial state may be set by using High-Low or 1-0 after the *Period*. *Pin* is automatically made an output.

Operators

Pin is a Port.Pin constant that specifies the I/O pin to use.

Period can be a constant of user defined variable. See notes.

State is an optional constant (0 or 1) or name **High - Low** that specifies the state of the outgoing pulse.

Example

```
' Send a high pulse 1ms long (at 4MHz) to PORTB Pin5
Low PORTB.5
PulseOut PORTB.5, 100

' Send a high pulse 1ms long (at 4MHz) to PORTB Pin5
PulseOut PORTB.5, 100, High
```

Notes

The resolution of **PulseOut** is dependent upon the oscillator frequency. If a 4MHz oscillator is used, the *Period* of the generated pulse will be in 10us increments. If a 20MHz oscillator is used, *Period* will have a 2us resolution. Declaring an Xtal value has no effect on **PulseOut**. The resolution always changes with the actual oscillator speed.

See also : Counter , PulseIn, RCin.

Push

Syntax

Push *Variable*, {*Variable*, *Variable* etc}

Overview

Place a single variable or multiple variables onto a software stack.

If the **Push** command is issued without a following variable, it will implement the assembler mnemonic **Push**, which manipulates the PICmicro's call stack.

Operators

Variable is a user defined variable of type **Bit**, **Byte**, **Word**, **Dword**, **Float**, **Array**, **String**, or **constant** value.

The amount of bytes pushed on to the stack varies with the variable type used. The list below shows how many bytes are pushed for a particular variable type, and their order.

Bit	1 Byte is pushed that holds the condition of the bit.
Byte	1 Byte is pushed.
Byte Array	1 Byte is pushed.
Word	2 Bytes are pushed. High Byte then Low Byte.
Word Array	2 Bytes are pushed. High Byte then Low Byte.
Dword Array	4 Bytes are pushed. High Byte then Low Byte.
Dword	4 Bytes are pushed. High Byte, Mid2 Byte, Mid1 Byte then Low Byte.
Float	4 Bytes are pushed. High Byte, Mid2 Byte, Mid1 Byte then Low Byte.
String	2 Bytes are pushed. High Byte then Low Byte that point to the start address of the string in memory.
Constant	Amount of bytes varies according to the value pushed. High Byte first.

Example 1

```
' Push two variables on to the stack then retrieve them

Device = 18F452      ' Stack only suitable for 18F devices
Declare Stack_Size = 20 ' Create a small stack capable of holding 20 bytes

Dim Wrld as Word    ' Create a Word variable
Dim Dwd as Dword    ' Create a Dword variable

Wrld = 1234          ' Load the Word variable with a value
Dwd = 567890        ' Load the Dword variable with a value
Push Wrld, Dwd      ' Push the Word variable then the Dword variable

Clear Wrld          ' Clear the Word variable
Clear Dwd           ' Clear the Dword variable

Pop Dwd, Wrld       ' Pop the Dword variable then the Word variable
Print Dec Wrld, " ", Dec Dwd ' Display the variables as decimal
Stop
```

Example 2

' Push a String on to the stack then retrieve it

```
Device = 18F452      ' Stack only suitable for 18F devices
Declare Stack_Size = 10 ' Create a small stack capable of holding 10 bytes

Dim SourceString as String * 20 ' Create a String variable
Dim DestString as String * 20  ' Create another String variable

SourceString = "HELLO WORLD"    ' Load the String variable with characters

Push SourceString      ' Push the String variable's address

Pop DestString ' Pop the previously pushed String into DestString
Print DestString ' Display the string, which will be "HELLO WORLD"
Stop
```

Formatting a Push.

Each variable type, and more so, constant value, will push a different amount of bytes on to the stack. This can be a problem where values are concerned because it will not be known what size variable is required in order to **Pop** the required amount of bytes from the stack. For example, the code below will push a constant value of 200 on to the stack, which requires 1 byte.

```
Push 200
```

All well and good, but what if the recipient popped variable is of a **Word** or **Dword** type.

```
Pop Wrd
```

Popping from the stack into a **Word** variable will actually pull 2 bytes from the stack, however, the code above has only pushed on byte, so the stack will become out of phase with the values or variables previously pushed. This is not really a problem where variables are concerned, as each variable has a known byte count and the user knows if a **Word** is pushed, a **Word** should be popped.

The answer lies in using a formatter preceding the value or variable pushed, that will force the amount of bytes loaded on to the stack. The formatters are **Byte**, **Word**, **Dword** or **Float**.

The **Byte** formatter will force any variable or value following it to push only 1 byte to the stack.

```
Push Byte 12345
```

The **Word** formatter will force any variable or value following it to push only 2 bytes to the stack:

```
Push Word 123
```

The **Dword** formatter will force any variable or value following it to push only 4 bytes to the stack: -

```
Push Dword 123
```

The **Float** formatter will force any variable or value following it to push only 4 bytes to the stack, and will convert a constant value into the 4-byte floating point format: -

```
Push Float 123
```

So for the **Push** of 200 code above, you would use: -

```
Push Word 200
```

In order for it to be popped back into a **Word** variable, because the push would be the high byte of 200, then the low byte.

If using the multiple variable **Push**, each parameter can have a different formatter preceding it.

```
Push Word 200, Dword 1234, Float 1234
```

Note that if a floating point value is pushed, 4 bytes will be placed on the stack because this is a known format.

What is a Stack?

All microprocessors and most microcontrollers have access to a Stack, which is an area of RAM allocated for temporary data storage. But this is sadly lacking on a PICmicro™ device. However, the 18F devices have an architecture and low-level mnemonics that allow a Stack to be created and used very efficiently.

A stack is first created in high memory by issuing the **Stack_Size Declare**.

```
Declare Stack_Size = 40
```

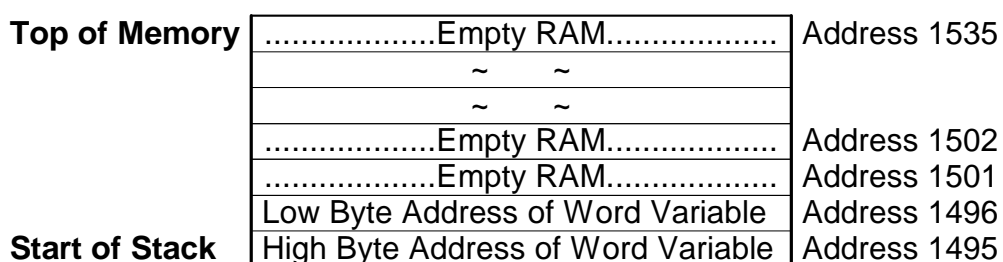
The above line of code will reserve 40 bytes at the top of RAM that cannot be touched by any BASIC command, other than **Push** and **Pop**. This means that it is a safe place for temporary variable storage.

Taking the above line of code as an example, we can examine what happens when a variable is pushed on to the 40 byte stack, and then popped off again.

First the RAM is allocated. For this explanation we will assume that a 18F452 PICmicro™ device is being used. The 18F452 has 1536 bytes of RAM that stretches linearly from address 0 to 1535. Reserving a stack of 40 bytes will reduce the top of memory so that the compiler will only see 1495 bytes (1535 - 40). This will ensure that it will not inadvertently try and use it for normal variable storage.

Pushing.

When a **Word** variable is pushed onto the stack, the memory map would look like the diagram below: -



The high byte of the variable is first pushed on to the stack, then the low byte. And as you can see, the stack grows in an upward direction whenever a **Push** is implemented, which means it shrinks back down whenever a **Pop** is implemented.

If we were to **Push** a **Dword** variable on to the stack as well as the **Word** variable, the stack memory would look like: -

Top of MemoryEmpty RAM.....	Address 1535
	~ ~	
	~ ~	
Empty RAM.....	Address 1502
Empty RAM.....	Address 1501
	Low Byte Address of Dword Variable	Address 1500
	Mid1 Byte Address of Dword Variable	Address 1499
	Mid2 Byte Address of Dword Variable	Address 1498
	High Byte Address of Dword Variable	Address 1497
	Low Byte Address of Word Variable	Address 1496
Start of Stack	High Byte Address of Word Variable	Address 1495

Popping.

When using the **Pop** command, the same variable type that was pushed last must be popped first, or the stack will become out of phase and any variables that are subsequently popped will contain invalid data. For example, using the above analogy, we need to **Pop** a **Dword** variable first. The **Dword** variable will be popped Low Byte first, then MID1 Byte, then MID2 Byte, then lastly the High Byte. This will ensure that the same value pushed will be reconstructed correctly when placed into its recipient variable. After the **Pop**, the stack memory map will look like: -

Top of MemoryEmpty RAM.....	Address 1535
	~ ~	
	~ ~	
Empty RAM.....	Address 1502
Empty RAM.....	Address 1501
	Low Byte Address of Word Variable	Address 1496
Start of Stack	High Byte Address of Word Variable	Address 1495

If a **Word** variable was then popped, the stack will be empty, however, what if we popped a **Byte** variable instead? the stack would contain the remnants of the **Word** variable previously pushed. Now what if we popped a **Dword** variable instead of the required **Word** variable? the stack would underflow by two bytes and corrupt any variables using those address's . The compiler cannot warn you of this occurring, so it is up to you, the programmer, to ensure that proper stack management is carried out. The same is true if the stack overflows. i.e. goes beyond the top of RAM. The compiler cannot give a warning.

Technical Details of Stack implementation.

The stack implemented by the compiler is known as an **Incrementing Last-In First-Out** Stack. *Incrementing* because it grows upwards in memory. *Last-In First-Out* because the last variable pushed, will be the first variable popped.

The stack is not circular in operation, so that a stack overflow will rollover into the PICmicro's hardware register, and an underflow will simply overwrite RAM immediately below the Start of Stack memory. If a circular operating stack is required, it will need to be coded in the main BASIC program, by examination and manipulation of the stack pointer (see below).

Indirect register pair FSR2L and FSR2H are used as a 16-bit stack pointer, and are incremented for every **Byte** pushed, and decremented for every **Byte** popped. Therefore checking the FSR2 registers in the BASIC program will give an indication of the stack's condition if required. This also means that the BASIC program cannot use the FSR2 register pair as part of its code, unless for manipulating the stack. Note that none of the compiler's commands, other than **Push** and **Pop**, use FSR2.

Whenever a variable is popped from the stack, the stack's memory is not actually cleared, only the stack pointer is moved. Therefore, the above diagrams are not quite true when they show empty RAM, but unless you have use of the remnants of the variable, it should be considered as empty, and will be overwritten by the next **Push** command.

See also : **Pop, Gosub, Return .**

Pwm

Syntax

Pwm *Pin, Duty, Cycles*

Overview

Output pulse-width-modulation on a pin, then return the pin to input state.

Operators

Pin is a Port.Pin constant that specifies the I/O pin to use.

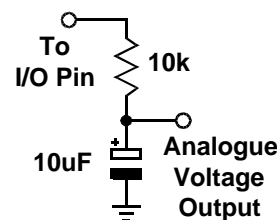
Duty is a variable, constant (0-255), or expression, which specifies the analogue level desired (0-5 volts).

Cycles is a variable or constant (0-255) which specifies the number of cycles to output. Larger capacitors require multiple cycles to fully charge. Cycle time is dependant on Xtal frequency. If a 4MHz crystal is used, then *cycle* takes approx 5 ms. If a 20MHz crystal is used, then *cycle* takes approx 1 ms.

Notes

Pwm can be used to generate analogue voltages (0-5V) through a pin connected to a resistor and capacitor to ground; the resistor-capacitor junction is the analogue output (see circuit). Since the capacitor gradually discharges, **Pwm** should be executed periodically to refresh the analogue voltage.

Pwm emits a burst of 1s and 0s whose ratio is proportional to the *duty* value you specify. If *duty* is 0, then the pin is continuously low (0); if *duty* is 255, then the pin is continuously high. For values in between, the proportion is $duty/255$. For example, if *duty* is 100, the ratio of 1s to 0s is $100/255 = 0.392$, approximately 39 percent.

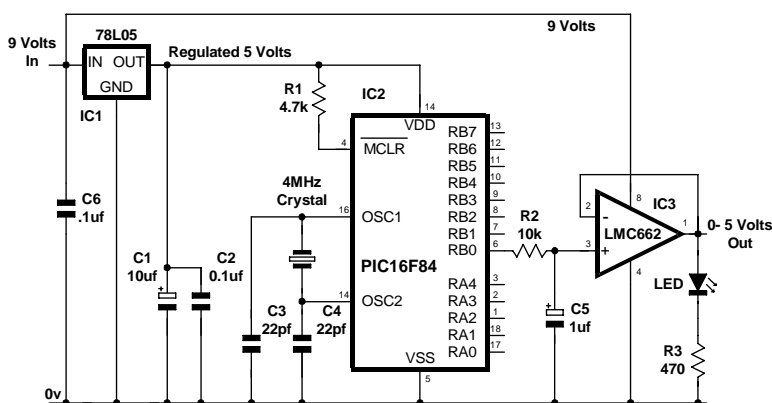


When such a burst is used to charge a capacitor arranged, the voltage across the capacitor is equal to:-

$$(duty / 255) * 5.$$

So if *duty* is 100, the capacitor voltage is

$$(100 / 255) * 5 = 1.96 \text{ volts.}$$



This voltage will drop as the capacitor discharges through whatever load it is driving. The rate of discharge is proportional to the current drawn by the load; more current = faster discharge. You can reduce this effect in software by refreshing the capacitor's charge with frequent use of the **Pwm** command, or you can buffer the output using an op-amp to greatly reduce the need for frequent **Pwm** cycles.

See also : **Hpwm, Pulseout, Servo.**

Random

Syntax

Variable = **Random**

or

Random *Variable*

Overview

Generate a pseudo-randomised value.

Operators

Variable is a user defined variable that will hold the pseudo-random value. The pseudo-random algorithm used has a working length of 1 to 65535 (only zero is not produced).

Example

```
Var1 = Random    ' Get a random number into Var1  
Random Var1     ' Get a random number into Var1
```

See also: **Seed.**

RC5in

Syntax

Variable = **RC5in**

Overview

Receive Philips RC5 infrared data from a predetermined pin. The pin is automatically made an input.

Operators

Variable can be a bit, byte, word, dword, or float variable, that will be loaded by **RC5in**. The return data from the **RC5in** command consists of two bytes, the System byte containing the type of remote used. i.e. TV, Video etc, and the Command byte containing the actual button value. The order of the bytes is Command (low byte) then System (high byte). If a byte variable is used to receive data from the infrared sensor then only the Command byte will be received.

Example

```
' Receive Philips RC5 data from an infrared sensor attached to PORTC.0
Device = 16F877
Declare RC5in_Pin = PORTC.0 ' Choose port.pin for infrared sensor
Dim RC5_Word as Word       ' Create a Word variable to receive the data
'
' Alias the Command byte to RC5_Word low byte
'
Dim RC5_Command as RC5_Word.Lowbyte
'
' Alias the System byte to RC5_Word high byte
'
Dim RC5_System as RC5_Word.Highbyte
ADCON1 = 7                ' PORTA and PORTE to all digital mode
Cls                        ' Clear the LCD
While                      ' Create an infinite loop
  Repeat
    RC5_Word = RC5In      ' Receive a signal from the infrared sensor
  Until RC5_Command <> 255 ' Keep looking until a valid header found
  Print at 1,1,"System  ",Dec RC5_System," " ' Display the System value
  Print at 2,1,"Command ",Dec RC5_Command," " ' Display the Command value
Wend
```

There is a single Declare for use with **RC5in**: -

Declare RC5in_Pin Port . Pin

Assigns the Port and Pin that will be used to input infrared data by the **RC5in** command. This may be any valid port on the PICmicro™.

If the **Declare** is not used in the program, then the default Port and Pin is PORTB.0.

Notes

The **RC5in** command will return with both Command and System bytes containing 255 if a valid header was not received. The CARRY (STATUS.0) flag will also be set if an invalid header was received. This is an ideal method of determining if the signal received is of the correct type.

RC5in is oscillator independent as long as the crystal frequency is declared at the top of the program. If no Xtal **Declare** is used, then **RC5in** defaults to a 4MHz crystal frequency for its timing.

RCin

Syntax

Variable = **RCin** *Pin*, *State*

Overview

Count time while pin remains in *state*, usually used to measure the charge/ discharge time of resistor/capacitor (RC) circuit.

Operators

Pin is a Port.Pin constant that specifies the I/O pin to use. This pin will be placed into input mode and left in that state when the instruction finishes.

State is a variable or constant (1 or 0) that will end the Rcin period. Text, High or Low may also be used instead of 1 or 0.

Variable is a variable in which the time measurement will be stored.

Example

```
Dim Result as Word           ' Word variable to hold result.
High PORTB.0                 ' Discharge the cap
DelayMs 1                     ' Wait for 1 ms.
Result = RCin PORTB.0, High  ' Measure RC charge time.
Print Dec Result, " "        ' Display the value on an LCD.
```

Notes

The resolution of **RCin** is dependent upon the oscillator frequency. If a 4MHz oscillator is used, the time in state is returned in 10us increments. If a 20MHz oscillator is used, the time in state will have a 2us resolution. Declaring an Xtal value has no effect on **RCin**. The resolution always changes with the actual oscillator speed. If the pin never changes state 0 is returned.

When **RCin** executes, it starts a counter. The counter stops as soon as the specified pin is no longer in *State* (0 or 1). If *pin* is not in *State* when the instruction executes, **RCin** will return 1 in *Variable*, since the instruction requires one timing cycle to discover this fact. If pin remains in *State* longer than 65535 timing cycles **RCin** returns 0.

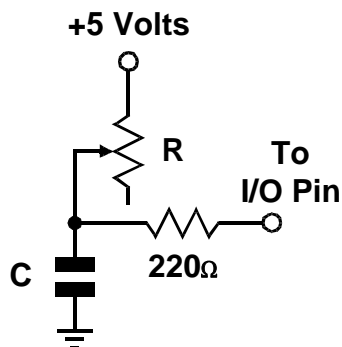


Figure A

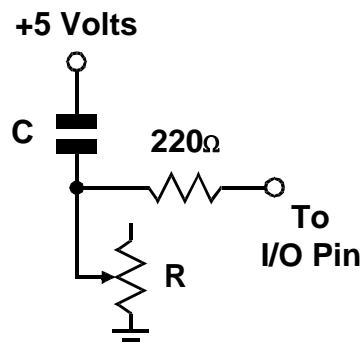


Figure B

The diagrams above show two suitable RC circuits for use with **RCin**. The circuit in figure B is preferred, because the PICmicro's logic threshold is approximately 1.5 volts. This means that the voltage seen by the pin will start at 5V then fall to 1.5V (a span of 3.5V) before **RCin** stops. With the circuit in figure A, the voltage will start at 0V and rise to 1.5V (spanning only 1.5V) before **RCin** stops.

For the same combination of R and C, the circuit shown in figure A will produce a higher result, and therefore more resolution than figure B.

Before **RCin** executes, the capacitor must be put into the state specified in the **RCin** command. For example, with figure B, the capacitor must be discharged until both plates (sides of the capacitor) are at 5V. It may seem strange that discharging the capacitor makes the input high, but you must remember that a capacitor is charged when there is a voltage difference between its plates. When both sides are at +5 Volts, the capacitor is considered discharged. Below is a typical sequence of instructions for the circuit in figure A.

```
Dim Result as Word           ' Word variable to hold result.
High PORTB.0                 ' Discharge the cap
DelayMs 1                     ' Wait for 1 ms.
Result = RCin PORTB.0, High  ' Measure RC charge time.
Print Dec Result, " "        ' Display the value on an LCD.
```

Using **RCin** is very straightforward, except for one detail: For a given R and C, what value will **RCin** return? It's actually rather easy to calculate, based on a value called the RC time constant, or tau (τ) for short. Tau represents the time required for a given RC combination to charge or discharge by 63 percent of the total change in voltage that they will undergo. More importantly, the value τ is used in the generalized RC timing calculation. Tau's formula is just R multiplied by C: -

$$\tau = R \times C$$

The general RC timing formula uses τ to tell us the time required for an RC circuit to change from one voltage to another: -

$$\text{time} = -\tau * (\ln(V_{\text{final}} / V_{\text{initial}}))$$

In this formula \ln is the natural logarithm. Assume we're interested in a 10k Ω resistor and 0.1 μ F cap. Calculate τ : -

$$\tau = (10 \times 10^3) \times (0.1 \times 10^{-6}) = 1 \times 10^{-3}$$

The RC time constant is 1×10^{-3} or 1 millisecond. Now calculate the time required for this RC circuit to go from 5V to 1.5V (as in figure B):

$$\text{Time} = -1 \times 10^{-3} * (\ln(5.0\text{v} / 1.5\text{v})) = 1.204 \times 10^{-3}$$

Using a 20MHz crystal, the unit of time is 2 μ s, that time (1.204×10^{-3}) works out to 602 units. With a 10k Ω resistor and 0.1 μ F capacitor, **RCin** would return a value of approximately 600. Since V_{initial} and V_{final} don't change, we can use a simplified rule of thumb to estimate **RCin** results for circuits similar to figure A: -

$$\text{RCin units} = 600 \times R (\text{in k}\Omega) \times C (\text{in }\mu\text{F})$$

Another useful rule of thumb can help calculate how long to charge/discharge the capacitor before **RCin**. In the example shown, that's the purpose of the **High** and **DelayMs** commands. A given RC charges or discharges 98 percent of the way in 4 time constants ($4 \times R \times C$).

In both circuits, the charge/discharge current passes through a 220 Ω series resistor and the capacitor. So if the capacitor were 0.1 μ F, the minimum charge/discharge time should be: -

Charge time = $4 \times 220 \times (0.1 \times 10^{-6}) = 88 \times 10^{-6}$

So it takes only 88 μ s for the cap to charge/discharge, which means that the 1ms charge/discharge time of the example is more than adequate.

You may be wondering why the 220 Ω resistor is necessary at all. Consider what would happen if resistor R in figure A were a pot, and was adjusted to 0 Ω . When the I/O pin went high to discharge the cap, it would see a short direct to ground. The 220 Ω series resistor would limit the short circuit current to $5V/220\Omega = 23mA$ and protect the PICmicro™ from any possible damage.

See also : Adin, Counter, Pot, PulseIn.

Repeat...Until

Syntax

Repeat *Condition*

Instructions

Instructions

Until *Condition*

or

Repeat { *Instructions* : } **Until** *Condition*

Overview

Execute a block of instructions until a condition is true.

Example

```
Var1 = 1
```

```
Repeat
```

```
    Print Dec Var1, " "
```

```
    DelayMs 200
```

```
    Inc Var1
```

```
Until Var1 > 10
```

or

```
Repeat High LED : Until PORTA.0 = 1 ' Wait for a Port change
```

Notes

The **Repeat-Until** loop differs from the **While-Wend** type in that, the **Repeat** loop will carry out the instructions within the loop at least once, then continuously until the condition is true, but the **While** loop only carries out the instructions if the condition is true.

The **Repeat-Until** loop is an ideal replacement to a **For-Next** loop, and actually takes less code space, thus performing the loop faster.

Two commands have been added especially for a **Repeat** loop, these are **Inc** and **Dec**.

Inc. Increment a variable i.e. `Var1 = Var1 + 1`

Dec. Decrement a variable i.e. `Var1 = Var1 - 1`

The above example shows the equivalent to the **For-Next** loop: -

```
For Var1 = 1 to 10 : Next
```

See also : **While...Wend, For...Next...Step.**

Resume

When the **Resume** statement is encountered at the end of the BASIC interrupt handler, it sets the GIE bit to re-enable interrupts and returns to where the program was before the interrupt occurred. Disable stops the compiler from inserting the Call to the interrupt checker before each command. This allows sections of code to execute without the possibility of being interrupted. Enable allows the insertion to continue.

A Disable should be placed before the interrupt handler so that it will not be restarted every time the GIE bit is checked. If it is desired to turn off interrupts for some reason after **On Interrupt** is encountered, you must not turn off the GIE bit. Turning off this bit informs the compiler an interrupt has happened and it will execute the interrupt handler forever.

Instead use: -

```
INTCON = $80
```

This disables all the individual interrupts but leaves the Global Interrupt Enable bit set.

A final note about interrupts in BASIC is if the program uses the command structure: -

```
Fin:  
  Goto Fin
```

You must remember the interrupt flag is checked before each instruction. It immediately jumps to label Fin with no interrupt check. Other commands must be placed in the loop for the interrupt check to happen: -

```
Fin:  
  DelayMs 1  
  Goto Fin
```

Note.

Software interrupts are a remnant from earlier compiler versions and are not recommended for new applications. See **Managed Hardware Interrupts** for a better method of interrupt handling.

See also : **Software Interrupts in BASIC, Disable, Enable.**

Return

Syntax Return

or

Return Variable

Availability

All devices. But a parameter return is only supported with 18F devices.

Overview

Return from a subroutine.

If using an 18F device, a parameter can be pushed onto a software stack before the return mnemonic is implemented.

Variable is a user defined variable of type **Bit**, **Byte**, **Word**, **Dword**, **Float**, **Array**, **String**, or **Constant** value, that will be pushed onto the stack before the subroutine is exited.

Example

```
' Call a subroutine with parameters
Device = 18F452      ' Stack only suitable for 18F devices
Declare Stack_size = 20 ' Create a small stack capable of holding 20 bytes

Dim Wrd1 as Word      ' Create a Word variable
Dim Wrd2 as Word      ' Create another Word variable
Dim Receipt as Word   ' Create a variable to hold result

Wrd1 = 1234           ' Load the Word variable with a value
Wrd2 = 567            ' Load the other Word variable with a value
' Call the subroutine and return a value
Gosub AddThem [Wrd1, Wrd2], Receipt
Print Dec Receipt    ' Display the result as decimal
Stop

' Subroutine starts here. Add two parameters passed and return the result
AddThem:
Dim AddWrd1 as Word  ' Create two uniquely named variables
Dim AddWrd2 as Word

Pop AddWrd2          ' Pop the last variable pushed
Pop AddWrd1          ' Pop the first variable pushed
AddWrd1 = AddWrd1 + AddWrd2    ' Add the values together
Return AddWrd1       ' Return the result of the addition
```

In reality, what's happening with the **Return** in the above program is simple, if we break it into its constituent events: -

```
Push AddWrd1  
Return
```

Notes

The same rules apply for the variable returned as they do for **Pop**, which is after all, what is happening when a variable is returned.

Return resumes execution at the statement following the **Gosub** which called the subroutine.

See also : **Call, Gosub, Push, Pop .**

Right\$

Syntax

Destination String = **Right\$** (*Source String*, *Amount of characters*)

Overview

Extract *n* amount of characters from the right of a source string and copy them into a destination string.

Overview

Destination String can only be a **String** variable, and should be large enough to hold the correct amount of characters extracted from the *Source String*.

Source String can be a **String** variable, or a **Quoted String of Characters**. See below for more variable types that can be used for *Source String*.

Amount of characters can be any valid variable type, expression or constant value, that signifies the amount of characters to extract from the right of the *Source String*. Values start at 1 for the rightmost part of the string and should not exceed 255 which is the maximum allowable length of a String variable.

Example 1

```
' Copy 5 characters from the right of SourceString into DestString

Device = 18F452           ' A suitable device for Strings
Dim SourceString as String * 20 ' Create a String of 20 characters
Dim DestString as String * 20  ' Create another String

SourceString = "HELLO WORLD" ' Load the source string with characters
' Copy 5 characters from the source string into the destination string
DestString = Right$(SourceString, 5)
Print DestString          ' Display the result, which will be "WORLD"
Stop
```

Example 2

```
' Copy 5 characters from right of a Quoted Character String to DestString

Device = 18F452           ' A suitable device for Strings
Dim DestString as String * 20 ' Create a String of 20 characters

' Copy 5 characters from the quoted string into the destination string
DestString = Right$("HELLO WORLD", 5)
Print DestString          ' Display the result, which will be "WORLD"
Stop
```

The *Source String* can also be a **Byte**, **Word**, **Dword**, **Float** or **Array**, variable, in which case the value contained within the variable is used as a pointer to the start of the Source String's address in RAM.

Example 3

```
' Copy 5 characters from the right of SourceString into DestString using a
' pointer to SourceString

Device = 18F452           ' A suitable device for Strings
Dim SourceString as String * 20 ' Create a String of 20 characters
Dim DestString as String * 20  ' Create another String
' Create a Word variable to hold the address of SourceString
Dim StringAddr as Word

SourceString = "HELLO WORLD" ' Load the source string with characters
' Locate the start address of SourceString in RAM
StringAddr = AddressOf(SourceString)
' Copy 5 characters from the source string into the destination string
DestString = Right$(StringAddr, 5)
Print DestString           ' Display the result, which will be "WORLD"
Stop
```

A third possibility for *Source String* is a Label name, in which case a null terminated Quoted String of Characters is read from a **Cdata** table.

Example 4

```
' Copy 5 characters from the right of a Cdata table into DestString

Device = 18F452           ' A suitable device for Strings
Dim DestString as String * 20 ' Create a String of 20 characters

' Copy 5 characters from label Source into the destination string
DestString = Right$(Source, 5)
Print DestString           ' Display the result, which will be "WORLD"
Stop

' Create a null terminated string of characters in code memory
Source:
Cdata "Hello World", 0
```

See also : **Creating and using Strings, Creating and using Virtual Strings with Cdata, Cdata, Len, Left\$, Mid\$, Str\$, ToLower, ToUpper, AddressOf.**

Rsin

Syntax

`Variable = Rsin, { Timeout Label }`

or

`Rsin { Timeout Label }, Modifier..Variable {, Modifier.. Variable...}`

Overview

Receive one or more bytes from a predetermined pin at a predetermined baud rate in standard asynchronous format using 8 data bits, no parity and 1 stop bit (8N1). The pin is automatically made an input.

Operators

Modifiers may be one of the serial data modifiers explained below.

Variable can be any user defined variable.

An optional *Timeout Label* may be included to allow the program to continue if a character is not received within a certain amount of time. *Timeout* is specified in units of 1 millisecond and is specified by using a **Declare** directive.

Example

```
Declare Rsin_Timeout = 2000      ' Timeout after 2 seconds
Dim Var1 as Byte
Dim Wrđ as Word
Var1 = Rsin, {Label}
Rsin Var1, Wrđ
Rsin { Label }, Var1, Wrđ
```

```
Label: { do something when timed out }
```

Declares

There are four **Declares** for use with **Rsin**. These are :-

Declare Rsin_Pin Port . Pin

Assigns the Port and Pin that will be used to input serial data by the **Rsin** command. This may be any valid port on the PICmicro™.

If the **Declare** is not used in the program, then the default Port and Pin is PORTB.1.

Declare Rsin_Mode_Inverted, True or 1, 0

Sets the serial mode for the data received by **Rsin**. This may be inverted or true. Alternatively, a value of 1 may be substituted to represent inverted, and 0 for true.

If the **Declare** is not used in the program, then the default mode is Inverted.

Declare Serial_Baud 0 to 65535 bps (baud)

Informs the **Rsin** and **Rsout** routines as to what baud rate to receive and transmit data.

Virtually any baud rate may be transmitted and received, but there are standard bauds: -

300, 600, 1200, 2400, 4800, 9600, and 19200.

When using a 4MHz crystal, the highest baud rate that is reliably achievable is 9600. However, an increase in the oscillator speed allows higher baud rates to be achieved, including 38400 baud.

If the **Declare** is not used in the program, then the default baud is 9600.

Declare Rsin_Timeout 0 to 65535 milliseconds (ms)

Sets the time, in milliseconds, that **Rsin** will wait for a start bit to occur.

Rsin waits in a tight loop for the presence of a start bit. If no timeout value is used, then it will wait forever. The **Rsin** command has the option of jumping out of the loop if no start bit is detected within the time allocated by timeout.

If the **Declare** is not used in the program, then the default timeout value is 10000ms or 10 seconds.

Rsin Modifiers.

As we already know, **Rsin** will wait for and receive a single byte of data, and store it in a variable . If the PICmicro™ were connected to a PC running a terminal program and the user pressed the "A" key on the keyboard, after the **Rsin** command executed, the variable would contain 65, which is the ASCII code for the letter "A"

What would happen if the user pressed the "1" key? The result would be that the variable would contain the value 49 (the ASCII code for the character "1"). This is an important point to remember: every time you press a character on the keyboard, the computer receives the ASCII value of that character. It is up to the receiving side to interpret the values as necessary. In this case, perhaps we actually wanted the variable to end up with the value 1, rather than the ASCII code 49.

The **Rsin** command provides a modifier, called the decimal modifier, which will interpret this for us. Look at the following code: -

```
Dim SerData as Byte
Rsin Dec SerData
```

Notice the decimal modifier in the **Rsin** command that appears just to the left of the SerData variable. This tells **Rsin** to convert incoming text representing decimal numbers into true decimal form and store the result in SerData. If the user running the terminal software pressed the "1", "2" and then "3" keys followed by a space or other non-numeric text, the value 123 will be stored in the variable SerData, allowing the rest of the program to perform any numeric operation on the variable.

Without the decimal modifier, however, you would have been forced to receive each character ("1", "2" and "3") separately, and then would still have to do some manual conversion to arrive at the number 123 (one hundred twenty three) before you can do the desired calculations on it.

The decimal modifier is designed to seek out text that represents decimal numbers. The characters that represent decimal numbers are the characters "0" through "9". Once the **Rsin** command is asked to use the decimal modifier for a particular variable, it monitors the incoming serial data, looking for the first decimal character. Once it finds the first decimal character, it will continue looking for more (accumulating the entire multi-digit number) until it finds a non-decimal numeric character. Remember that it will not finish until it finds at least one decimal character followed by at least one non-decimal character.

To illustrate this further, examine the following examples (assuming we're using the same code example as above): -

Serial input: "ABC"

Result: The program halts at the **Rsin** command, continuously waiting for decimal text.

Serial input: "123" (with no characters following it)

Result: The program halts at the **Rsin** command. It recognises the characters "1", "2" and "3" as the number one hundred twenty three, but since no characters follow the "3", it waits continuously, since there's no way to tell whether 123 is the entire number or not.

Serial input: "123" (followed by a space character)

Result: Similar to the above example, except once the space character is received, the program knows the entire number is 123, and stores this value in SerData. The **Rsin** command then ends, allowing the next line of code to run.

Serial input: "123A"

Result: Same as the example above. The "A" character, just like the space character, is the first non-decimal text after the number 123, indicating to the program that it has received the entire number.

Serial input: "ABCD123EFGH"

Result: Similar to examples 3 and 4 above. The characters "ABCD" are ignored (since they're not decimal text), the characters "123" are evaluated to be the number 123 and the following character, "E", indicates to the program that it has received the entire number.

The final result of the **Dec** modifier is limited to 16 bits (up to the value 65535). If a value larger than this is received by the decimal modifier, the end result will be incorrect because the result rolled-over the maximum 16-bit value. Therefore, **Rsin** modifiers may not (at this time) be used to load **Dword** (32-bit) variables.

The decimal modifier is only one of a family of conversion modifiers available with **Rsin**. See below for a list of available conversion modifiers. All of the conversion modifiers work similar to the decimal modifier (as described above). The modifiers receive bytes of data, waiting for the first byte that falls within the range of characters they accept (e.g., "0" or "1" for binary, "0" to "9" for decimal, "0" to "9" and "A" to "F" for hex). Once they receive a numeric character, they keep accepting input until a non-numeric character arrives, or in the case of the fixed length modifiers, the maximum specified number of digits arrives.

While very effective at filtering and converting input text, the modifiers aren't completely fool-proof. As mentioned before, many conversion modifiers will keep accepting text until the first non-numeric text arrives, even if the resulting value exceeds the size of the variable. After **Rsin**, a **Byte** variable will contain the lowest 8 bits of the value entered and a **Word** (16-bits) would contain the lowest 16 bits. You can control this to some degree by using a modifier that specifies the number of digits, such as **Dec2**, which would accept values only in the range of 0 to 99.

Conversion Modifier	Type of Number	Numeric	Characters Accepted
Dec {1..10}	Decimal, optionally limited		0 through 9
	to 1 - 10 digits		
Hex {1..8}	Hexadecimal, optionally limited		0 through 9,
	to 1 - 8 digits		A through F
Bin {1..32}	Binary, optionally limited		0, 1
	to 1 - 32 digits		

A variable preceded by **Bin** will receive the ASCII representation of its binary value. For example, if **Bin** Var1 is specified and "1000" is received, Var1 will be set to 8.

A variable preceded by **Dec** will receive the ASCII representation of its decimal value. For example, if **Dec** Var1 is specified and "123" is received, Var1 will be set to 123.

A variable preceded by **Hex** will receive the ASCII representation of its hexadecimal value. For example, if **Hex** Var1 is specified and "FE" is received, Var1 will be set to 254.

SKIP followed by a count will skip that many characters in the input stream. For example, **SKIP** 4 will skip 4 characters.

The **Rsin** command can be configured to wait for a specified sequence of characters before it retrieves any additional input. For example, suppose a device attached to the PICmicro™ is known to send many different sequences of data, but the only data you wish to observe happens to appear right after the unique characters, "XYZ". A modifier named **Wait** can be used for this purpose: -

```
Rsin Wait("XYZ"), SerData
```

The above code waits for the characters "X", "Y" and "Z" to be received, in that order, then it receives the next data byte and places it into variable SerData.

Str modifier.

The **Rsin** command also has a modifier for handling a string of characters, named **Str**.

The **Str** modifier is used for receiving a string of characters into a byte array variable.

A string is a set of characters that are arranged or accessed in a certain order. The characters "ABC" would be stored in a string with the "A" first, followed by the "B" then followed by the "C". A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string "ABC" would be stored in a byte array containing three bytes (elements).

Below is an example that receives ten bytes and stores them in the 10-byte array, SerString: -

```
Dim SerString[10] as Byte    ' Create a 10-byte array.
Rsin Str SerString          ' Fill the array with received data.
Print Str SerString         ' Display the string.
```

If the amount of received characters is not enough to fill the entire array, then a formatter may be placed after the array's name, which will only receive characters until the specified length is reached. For example: -

```
Dim SerString[10] as Byte    ' Create a 10-byte array.
Rsin Str SerString\5         ' Fill the first 5-bytes of the array
Print Str SerString\5       ' Display the 5-character string.
```

The example above illustrates how to fill only the first *n* bytes of an array, and then how to display only the first *n* bytes of the array. *n* refers to the value placed after the backslash.

Because of its complexity, serial communication can be rather difficult to work with at times. Using the guidelines below when developing a project using the **Rsin** and **Rsout** commands may help to eliminate some obvious errors: -

Always build your project in steps.

Start with small, manageable pieces of code, (that deal with serial communication) and test them, one individually.

Add more and more small pieces, testing them each time, as you go.

Never write a large portion of code that works with serial communication without testing its smallest workable pieces first.

Pay attention to timing.

Be careful to calculate and overestimate the amount of time, operations should take within the PICmicro™ for a given oscillator frequency. Misunderstanding the timing constraints is the source of most problems with code that communicate serially. If the serial communication in your project is bi-directional, the above statement is even more critical.

Pay attention to wiring.

Take extra time to study and verify serial communication wiring diagrams. A mistake in wiring can cause strange problems in communication, or no communication at all. Make sure to connect the ground pins (Vss) between the devices that are communicating serially.

Verify port setting on the PC and in the Rsin / Rsout commands.

Unmatched settings on the sender and receiver side will cause garbled data transfers or no data transfers. This is never more critical than when a line transceiver is used (i.e. MAX232). Always remember that a line transceiver inverts the serial polarity.

If the serial data received is unreadable, it is most likely caused by a baud rate setting error, or a polarity error.

If receiving data from another device that is not a PICmicro™, try to use baud rates of 9600 and below, or alternatively, use a higher frequency crystal.

Because of additional overheads in the PICmicro™, and the fact that the **Rsin** command offers no hardware receive buffer for serial communication, received data may sometimes be missed or garbled. If this occurs, try lowering the baud rate, or increasing the crystal frequency. Using simple variables (not arrays) will also increase the chance that the PICmicro™ will receive the data properly.

Notes

Rsin is oscillator independent as long as the crystal frequency is declared at the top of the program. If no **Xtal Declare** is used, then **Rsin** defaults to a 4MHz crystal frequency for its bit timing.

See also : **Declare, Rsout, Serin, Serout, Hrsin, Hrsout, Hserin, Hserout.**

Rsout

Syntax

Rsout *Item* {, *Item...* }

Overview

Send one or more *Items* to a predetermined pin at a predetermined baud rate in standard asynchronous format using 8 data bits, no parity and 1 stop bit (8N1). The pin is automatically made an output.

Operators

Item may be a constant, variable, expression, or string list.

There are no operators as such, instead there are *modifiers*. For example, if an at sign '@' precedes an *Item*, the ASCII representation for each digit is transmitted.

The modifiers are listed below: -

Modifier	Operation
At ypos,xpos	Position the cursor on a serial LCD
ClS	Clear a serial LCD (also creates a 30ms delay)
Bin{1..32}	Send binary digits
Dec{1..10}	Send decimal digits
Hex{1..8}	Send hexadecimal digits
Sbin{1..32}	Send signed binary digits
Sdec{1..10}	Send signed decimal digits
Shex{1..8}	Send signed hexadecimal digits
Ibin{1..32}	Send binary digits with a preceding '%' identifier
Idec{1..10}	Send decimal digits with a preceding '#' identifier
Ihex{1..8}	Send hexadecimal digits with a preceding '\$' identifier
ISbin{1..32}	Send signed binary digits with a preceding '%' identifier
ISdec{1..10}	Send signed decimal digits with a preceding '#' identifier
IShex{1..8}	Send signed hexadecimal digits with a preceding '\$' identifier
Rep c\n	Send character c repeated n times
Str array\n	Send all or part of an array
Cstr cdata	Send string data defined in a Cdata statement.

The numbers after the **Bin**, **Dec**, and **Hex** modifiers are optional. If they are omitted, then the default is all the digits that make up the value will be displayed.

If a floating point variable is to be displayed, then the digits after the **Dec** modifier determine how many remainder digits are send. i.e. numbers after the decimal point.

```
Dim Flt as Float
Flt = 3.145
Rsout Dec2 Flt    ' Send 2 values after the decimal point
```

The above program will send 3.14

If the digit after the **Dec** modifier is omitted, then 3 values will be displayed after the decimal point.

```
Dim Flt as Float
Flt = 3.1456
Rsout Dec Flt      ' Send 3 values after the decimal point
```

The above program will send 3.145

There is no need to use the **Sdec** modifier for signed floating point values, as the compiler's **Dec** modifier will automatically display a minus result: -

```
Dim Flt as Float
Flt = -3.1456
Rsout Dec Flt      ' Send 3 values after the decimal point
```

The above program will send -3.145

Hex or **Bin** modifiers cannot be used with floating point values or variables.

The Xpos and Ypos values in the **At** modifier both start at 1. For example, to place the text "HELLO WORLD" on line 1, position 1, the code would be: -

```
Rsout At 1, 1, "HELLO WORLD"
```

Example 1

```
Dim Var1 as Byte
Dim Wrđ as Word
Dim Dwd as Dword

Rsout "Hello World"      ' Display the text "Hello World"
Rsout "Var1= ", Dec Var1 ' Display the decimal value of Var1
Rsout "Var1= ", Hex Var1 ' Display the hexadecimal value of Var1
Rsout "Var1= ", Bin Var1 ' Display the binary value of Var1
Rsout "Dwd= ", Hex6 Dwd  ' Display 6 hex characters of a Dword variable
```

Example 2

```
' Display a negative value on a serial LCD.
Symbol Negative = -200
Rsout At 1, 1, Sdec Negative
```

Example 3

```
' Display a negative value on a serial LCD with a preceding identifier.
Rsout At 1, 1, IShex -$1234
```

Example 3 will produce the text "\$-1234" on the LCD.

Some PICmicros such as the 16F87x, and 18FXXX range have the ability to read and write to their own flash memory. And although writing to this memory too many times is unhealthy for the PICmicro™, reading this memory is both fast, and harmless. Which offers a unique form of data storage and retrieval, the **Cdata** command proves this, as it uses the mechanism of reading and storing in the PICmicro's flash memory.

Combining the unique features of the 'self modifying PICmicro's' with a string format, the compiler is capable of reducing the overhead of printing, or transmitting large amounts of text data.

The **Cstr** modifier may be used in commands that deal with text processing i.e. **Serout**, **Hrsout**, and **Print** etc.

The **Cstr** modifier is used in conjunction with the **Cdata** command. The **Cdata** command is used for initially creating the string of characters: -

```
String1: Cdata "HELLO WORLD", 0
```

The above line of code will create, in flash memory, the values that make up the ASCII text "HELLO WORLD", at address String1. Note the null terminator after the ASCII text.

null terminated means that a zero (null) is placed at the end of the string of ASCII characters to signal that the string has finished.

To display, or transmit this string of characters, the following command structure could be used:

```
Rsout Cstr String1
```

The label that declared the address where the list of Cdata values resided, now becomes the string's name. In a large program with lots of text formatting, this type of structure can save quite literally hundreds of bytes of valuable code space.

Try both these small programs, and you'll see that using **Cstr** saves a few bytes of code: -

First the standard way of displaying text: -

```
Device = 16F877  
Cls  
Rsout "HELLO WORLD"  
Rsout "HOW ARE YOU?"  
Rsout "I AM FINE!"  
Stop
```

Now using the **Cstr** modifier: -

```
Cls  
Rsout Cstr TEXT1  
Rsout Cstr TEXT2  
Rsout Cstr TEXT3  
Stop
```

```
TEXT1: Cdata "HELLO WORLD", 13, 0  
TEXT2: Cdata "HOW ARE YOU?", 13, 0  
TEXT3: Cdata "I AM FINE!", 13, 0
```

Again, note the null terminators after the ASCII text in the Cdata commands. Without these, the PICmicro™ will continue to transmit data in an endless loop.

The term 'virtual string' relates to the fact that a string formed from the Cdata command cannot be written too, but only read from.

The **Str** modifier is used for sending a string of bytes from a byte array variable. A string is a set of bytes sized values that are arranged or accessed in a certain order.

The values 1, 2, 3 would be stored in a string with the value 1 first, followed by 2 then followed by the value 3. A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string 1,2,3 would be stored in a byte array containing three bytes (elements).

Below is an example that displays four bytes (from a byte array): -

```
Dim MyArray[10] as Byte      ' Create a 10-byte array.
MyArray [0] = "H"           ' Load the first 5 bytes of the array
MyArray [1] = "E"           ' With the data to send
MyArray [2] = "L"
MyArray [3] = "L"
MyArray [4] = "O"
Rsout Str MyArray\5         ' Display a 5-byte string.
```

Note that we use the optional \n argument of **Str**. If we didn't specify this, the PICmicro™ would try to keep sending characters until all 10 bytes of the array were transmitted. Since we do not wish all 10 bytes to be transmitted, we chose to tell it explicitly to only send the first 5 bytes.

The above example may also be written as: -

```
Dim MyArray[10] as Byte      ' Create a 10-byte array.
Str MyArray = "HELLO"        ' Load the first 5 bytes of the array
Rsout Str MyArray\5         ' Send 5-byte string.
```

The above example, has exactly the same function as the previous one. The only difference is that the string is now constructed using **Str** as a command instead of a modifier.

Declares

There are four **Declares** for use with **Rsout**. These are : -

Declare Rsout_Pin Port . Pin

Assigns the Port and Pin that will be used to output serial data from the **Rsout** command. This may be any valid port on the PICmicro™.

If the **Declare** is not used in the program, then the default Port and Pin is PORTB.0.

Declare Rsout_Mode_Inverted, True/False or 1, 0

Sets the serial mode for the data transmitted by **Rsout**. This may be inverted or true. Alternatively, a value of 1 may be substituted to represent inverted, and 0 for true.

If the **Declare** is not used in the program, then the default mode is INVERTED.

Declare Serial_Baud 0 to 65535 bps (baud)

Informs the **Rsin** and **Rsout** routines as to what baud rate to receive and transmit data.

Virtually any baud rate may be transmitted and received, but there are standard bauds: -

300, 600, 1200, 2400, 4800, 9600, and 19200.

When using a 4MHz crystal, the highest baud rate that is reliably achievable is 9600. However, an increase in the oscillator speed allows higher baud rates to be achieved, including 38400 baud.

If the **Declare** is not used in the program, then the default baud is 9600.

Declare Rsout_Pace 0 to 65535 microseconds (us)

Implements a delay between characters transmitted by the **Rsout** command.

On occasion, the characters transmitted serially are in a stream that is too fast for the receiver to catch, this results in missed characters. To alleviate this, a delay may be implemented between each individual character transmitted by **Rsout**.

If the **Declare** is not used in the program, then the default is no delay between characters.

Notes

Rsout is oscillator independent as long as the crystal frequency is declared at the top of the program. If no declare is used, then **Rsout** defaults to a 4MHz crystal frequency for its bit timing.

The **At** and **Cls** modifiers are primarily intended for use with serial LCD modules. Using the following command sequence will first clear the LCD, then display text at position 5 of line 2: -

```
Rsout Cls, At 2, 5, "HELLO WORLD"
```

The values after the **At** modifier may also be variables.

See also : **Declare, Rsin , Serin, Serout, Hrsin, Hrsout, Hserin, Hserout.**

Seed

Syntax

Seed *Value*

Overview

Seed the random number generator, in order to obtain a more random result.

Operators

Value can be a variable, constant or expression, with a value from 1 to 65535. A value of \$0345 is a good starting point.

Example

```
' Create and display a Random number
Device = 16F877
Declare Xtal = 4
Dim Rnd as Word

Seed $0345
Cls
Again:
  Rnd = Random
  Print At 1,1,Dec Rnd, "      "
  DelayMs 500
  Goto Again
```

See also: [Random.](#)

Select..Case..EndSelect

Syntax

Select *Expression*

Case *Condition(s)*
 Instructions

{
 Case *Condition(s)*
 Instructions

Case Else
 Statement(s)

}

EndSelect

The curly braces signify optional conditions.

Overview

Evaluate an *Expression* then continually execute a block of BASIC code based upon comparisons to *Condition(s)*. After executing a block of code, the program continues at the line following the **EndSelect**. If no conditions are found to be True and a **Case Else** block is included, the code after the **Case Else** leading to the **EndSelect** will be executed.

Operators

Expression can be any valid variable, constant, expression or inline command that will be compared to the *Conditions*.

Condition(s) is a statement that can evaluate as True or False. The Condition can be a simple or complex relationship, as described below. Multiple conditions within the same **Case** can be separated by commas.

Instructions can be any valid BASIC command that will be operated on if the **Case** condition produces a True result.

Example

```

' Load variable Result according to the contents of variable Var1
' Result will return a value of 255 if no valid condition was met
Include "Proton_4.Inc" ' Use the Proton development board for the demo
Dim Var1 as Byte
Dim Result as Byte
DelayMs 100          ' Wait for the LCD to stabilise
Cls                  ' Clear the LCD
Result = 0           ' Clear the result variable before we start
Var1 = 1              ' Variable to base the conditions upon
Select Var1
  Case 1              ' Is Var1 equal to 1 ?
    Result = 1        ' Load Result with 1 if yes
  Case 2              ' Is Var1 equal to 2 ?
    Result = 2        ' Load Result with 2 if yes
  Case 3              ' Is Var1 equal to 3 ?
    Result = 3        ' Load Result with 3 if yes
  Case Else          ' Otherwise...
    Result = 255      ' Load Result with 255
EndSelect
Print Dec Result    ' Display the result
Stop

```

Notes

Select..Case is simply an advanced form of the **If..Then..Elseif..Else** construct, in which multiple **Elseif** statements are executed by the use of the **Case** command.

Taking a closer look at the **Case** command: -

Case *Conditional_Op Expression*

Where *Conditional_Op* can be an = operator (which is implied if absent), or one of the standard comparison operators <>, <, >, >= or <=. Multiple conditions within the same **Case** can be separated by commas. If, for example, you wanted to run a **Case** block based on a value being less than one or greater than nine, the syntax would look like: -

```
Case < 1, > 9
```

Another way to implement Case is: -

```
Case value1 to value2
```

In this form, the valid range is from *Value1* to *Value2*, inclusive. So if you wished to run a Case block on a value being between the values 1 and 9 inclusive, the syntax would look like: -

```
Case 1 to 9
```

For those of you that are familiar with C or Java, you will know that in those languages the statements in a **Case** block fall through to the next **Case** block unless the keyword break is encountered. In BASIC however, the code under an executed **Case** block jumps to the code immediately after **EndSelect**.

Shown below is a typical **Select...Case** structure with its corresponding If..Then equivalent code alongside.

```
Select Var1
  Case 6, 9, 99, 66
    ' If Var1 = 6 or Var1 = 9 or Var1 = 99 or Var1 = 66 Then
      Print "or ValueS"
  Case 110 to 200
    ' ElseIf Var1 >= 110 and Var1 <= 200 Then
      Print "and ValueS"
  Case 100
    ' ElseIf Var1 = 100 Then
      Print "EQUAL Value"
  Case > 300
    ' ElseIf Var1 > 300 Then
      Print "GREATER Value"
  Case Else
    ' Else
      Print "DEFAULT Value"
EndSelect
' EndIf
```

See also : **If..Then..Elseif..Else..EndIf.**

Serin

Syntax

Serin *Rpin* { \ *Fpin* }, *Baudmode*, { *Plabel*, } { *Timeout*, *Tlabel*, } [*InputData*]

Overview

Receive asynchronous serial data (i.e. RS232 data).

Operators

Rpin is a Port.Bit constant that specifies the I/O pin through which the serial data will be received. This pin will be set to input mode.

Fpin is an optional Port.Bit constant that specifies the I/O pin to indicate flow control status on. This pin will be set to output mode.

Baudmode may be a variable, constant, or expression (0 - 65535) that specifies serial timing and configuration.

Plabel is an optional label indicating where the program should jump to in the event of a parity error. This argument should only be provided if **Baudmode** indicates that parity is required.

Timeout is an optional constant (0 - 65535) that informs **Serin** how long to wait for incoming data. If data does not arrive in time, the program will jump to the address specified by **Tlabel**.

Tlabel is an optional label that must be provided along with **Timeout**, indicating where the program should go in the event that data does not arrive within the period specified by **Timeout**.

InputData is list of variables and modifiers that informs **Serin** what to do with incoming data.

Serin may store data in a variable, array, or an array string using the **Str** modifier.

Notes

One of the most popular forms of communication between electronic devices is serial communication. There are two major types of serial communication; asynchronous and synchronous. The **Rsin**, **Rout**, **Serin** and **Serout** commands are all used to send and receive asynchronous serial data. While the **Shin** and **Shout** commands are for use with synchronous communications.

The term asynchronous means 'no clock.' More specifically, 'asynchronous serial communication' means data is transmitted and received without the use of a separate 'clock' line. Data can be sent using as few as two wires; one for data and one for ground. The PC's serial ports (also called COM ports or RS232 ports) use asynchronous serial communication. Note: the other kind of serial communication, synchronous, uses at least three wires; one for clock, one for data and one for ground.

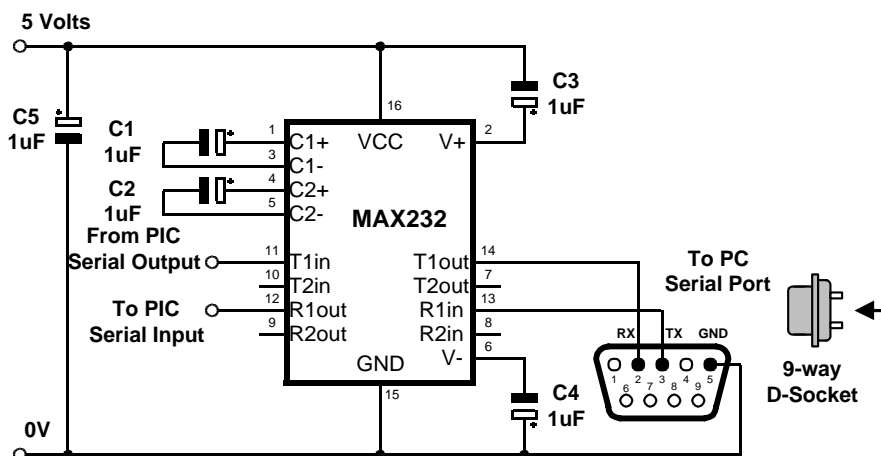
RS232 is the electrical specification for the signals that PC serial ports use. Unlike standard TTL logic, where 5 volts is a logic 1 and 0 volts is logic 0, RS232 uses -12 volts for logic 1 and +12 volts for logic 0. This specification allows communication over longer wire lengths without amplification.

Most circuits that work with RS232 use a line driver / receiver (transceiver). This component does two things: -

- Convert the ± 12 volts of RS-232 to TTL compatible 0 to 5 volt levels.
- Invert the voltage levels, so that 5 volts = logic 1 and 0 volts = logic 0.

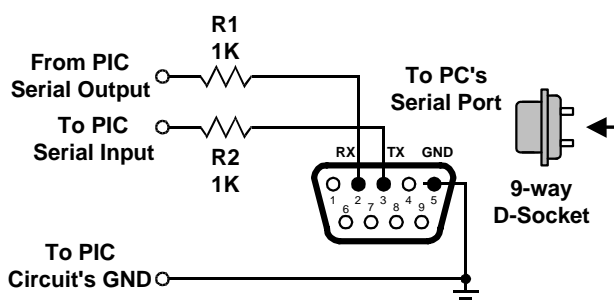
By far, the most common line driver device is the MAX232 from Maxim semiconductor. With the addition of a few capacitors, a complete 2-way level converter is realised. Figure 1 shows a typical circuit for one of these devices. The MAX232 is not the only device available, there are

other types that do not require any external capacitors at all. Visit Maxim's excellent web site at www.maxim.com, and download one of their many detailed datasheets.



Typical MAX232 RS232 line-transceiver circuit.

Because of the excellent IO capabilities of the PICmicro™ range of devices, and the adoption of TTL levels on most modern PC serial ports, a line driver is often unnecessary unless long distances are involved between the transmitter and the receiver. Instead a simple current limiting resistor is all that's required. As shown below: -



Directly connected RS232 circuit.

You should remember that when using a line transceiver such as the MAX232, the serial mode (polarity) is inverted in the process of converting the signal levels, however, if using the direct connection, the mode is untouched. This is the single most common cause of errors when connecting serial devices, therefore you must make allowances for this within your software.

Asynchronous serial communication relies on precise timing. Both the sender and receiver must be set for identical timing, this is commonly expressed in bits per second (bps) called baud. **SerIn** requires a value called *Baudmode* that informs it of the relevant characteristics of the incoming serial data; the bit period, number of data and parity bits, and polarity.

The *Baudmode* argument for **SerIn** accepts a 16-bit value that determines its characteristics: 1-stop bit, 8-data bits/no-parity or 7-data bits/even-parity and most speeds from as low as 300 baud to 38400 baud (depending on the crystal frequency used). The following table shows how *Baudmode* is calculated, while table 1 shows some common *baudmodes* for standard serial baud rates.

Step 1.	Determine the bit period. (bits 0 – 11)	$(1,000,000 / \text{baud rate}) - 20$
Step 2.	data bits and parity. (bit 13)	8-bit/no-parity = step 1 + 0 7-bit/even-parity = step 1 + 8192
Step 3.	Select polarity. (bit 14)	True (noninverted) = step 2 + 0 Inverted = step 2 + 16384

Baudmode calculation.

Add the results of steps 1, 2 3, and 3 to determine the correct value for the *Baudmode* operator.

BaudRate	8-bit no-parity inverted	8-bit no-parity true	7-bit even-parity inverted	7-bit even-parity true
300	19697	3313	27889	11505
600	18030	1646	26222	9838
1200	17197	813	25389	9005
2400	16780	396	24972	8588
4800	16572	188	24764	8380
9600	16468	84	24660	8276

Table 1. Common baud rates and corresponding *Baudmodes*.

If communications are with existing software or hardware, its speed and mode will determine the choice of baud rate and mode. In general, 7-bit/even-parity (7E) mode is used for text, and 8-bit/no-parity (8N) for byte-oriented data. Note: the most common mode is 8-bit/no-parity, even when the data transmitted is just text. Most devices that use a 7-bit data mode do so in order to take advantage of the parity feature. Parity can detect some communication errors, but to use it you lose one data bit. This means that incoming data bytes transferred in 7E (even-parity) mode can only represent values from 0 to 127, rather than the 0 to 255 of 8N (no-parity) mode.

The compiler's serial commands **Serin** and **Serout**, have the option of still using a parity bit with 4 to 8 data bits. This is through the use of a **Declare**: -

With parity disabled (the default setting): -

```
Declare Serial_Data 4 ' Set Serin and Serout data bits to 4
Declare Serial_Data 5 ' Set Serin and Serout data bits to 5
Declare Serial_Data 6 ' Set Serin and Serout data bits to 6
Declare Serial_Data 7 ' Set Serin and Serout data bits to 7
Declare Serial_Data 8 ' Set Serin and Serout data bits to 8 (default)
```

With parity enabled: -

```
Declare Serial_Data 5 ' Set Serin and Serout data bits to 4
Declare Serial_Data 6 ' Set Serin and Serout data bits to 5
Declare Serial_Data 7 ' Set Serin and Serout data bits to 6
Declare Serial_Data 8 ' Set Serin and Serout data bits to 7 (default)
Declare Serial_Data 9 ' Set Serin and Serout data bits to 8
```

Serial_Data data bits may range from 4 bits to 8 (the default if no **Declare** is issued). Enabling parity uses one of the number of bits specified.

Declaring **Serial_Data** as 9 allows 8 bits to be read and written along with a 9th parity bit.

Parity is a simple error-checking feature. When a serial sender is set for even parity (the mode the compiler supports) it counts the number of 1s in an outgoing byte and uses the parity bit to make that number even. For example, if it is sending the 7-bit value: %0011010, it sets the parity bit to 1 in order to make an even number of 1s (four).

The receiver also counts the data bits to calculate what the parity bit should be. If it matches the parity bit received, the serial receiver assumes that the data was received correctly. Of course, this is not necessarily true, since two incorrectly received bits could make parity seem correct when the data was wrong, or the parity bit itself could be bad when the rest of the data was correct.

Many systems that work exclusively with text use 7-bit/ even-parity mode. For example, to receive one data byte through bit-0 of PORTA at 9600 baud, 7E, inverted:

```
Serin PORTA.0, 24660, [SerData]
```

The above example will work correctly, however it doesn't inform the program what to do in the event of a parity error.

Below, is an improved version that uses the optional *Plabel* argument:

```
Serin PORTA.0, 24660, ParityError, [SerData]
Print Dec SerData
Stop
ParityError:
Print "Parity Error"
Stop
```

If the parity matches, the program continues at the **Print** instruction after **Serin**. If the parity doesn't match, the program jumps to the label P_ERROR. Note that a parity error takes precedence over other *InputData* specifications (as soon as an error is detected, **Serin** aborts and jumps to the *Plabel* routine).

In the examples above, the only way to end the **Serin** instruction (other than RESet or power-off) is to give **Serin** the serial data it needs. If no serial data arrives, the program is stuck in an endless loop. However, you can force **Serin** to abort if it doesn't receive data within a specified number of milliseconds.

For example, to receive a value through bit-0 of PORTA at 9600 baud, 8N, inverted and abort **Serin** after 2 seconds (2000 ms) if no data arrives: -

```
Serin PORTA.0, 16468, 2000, TimeoutError, [SerData]
Print Cls, Dec Result
Stop
TimeoutError:
Print Cls, "Timed Out"
Stop
```

If no serial data arrives within 2 seconds, **Serin** aborts and continues at the label to_ERROR.

Both Parity and Serial Timeouts may be combined. Below is an example to receive a value through bit-0 of PORTA at 2400 baud, 7E, inverted with a 10-second timeout: -

```

Dim SerData as Byte
Again:
  Serin PORTA.0, 24660, ParityError, 10000, TimeoutError, [SerData]
  Print Cls, Dec SerData
  Goto Again
TimeoutError:
  Print Cls, "Timed Out"
  Goto Again
ParityError:
  Print Cls, "Parity Error"
  Goto Again

```

When designing an application that requires serial communication between PICs, you should remember to work within these limitations: -

When the PICmicro™ is sending or receiving data, it cannot execute other instructions. When the PICmicro™ is executing other instructions, it cannot send or receive data. The compiler does not offer a serial buffer as there is in PCs. At lower crystal frequencies, and higher serial rates, the PICmicro™ cannot receive data via **Serin**, process it, and execute another **Serin** in time to catch the next chunk of data, unless there are significant pauses between data transmissions.

These limitations can sometimes be addressed by using flow control; the *Fpin* option for **Serin** and **Serout**. Through *Fpin*, **Serin** can inform another PICmicro™ sender when it is ready to receive data. (*Fpin* flow control follows the rules of other serial handshaking schemes, however most computers other than the PICmicro™ cannot start and stop serial transmission on a byte-by-byte basis. That is why this discussion is limited to communication between PICmicros.)

Below is an example using flow control with data through bit-0 of PORTA, and flow control through bit-1 of PORTA, 9600 baud, N8, noninverted: -

```

Serin PORTA.0\PORTA.1, 84, [SerData]

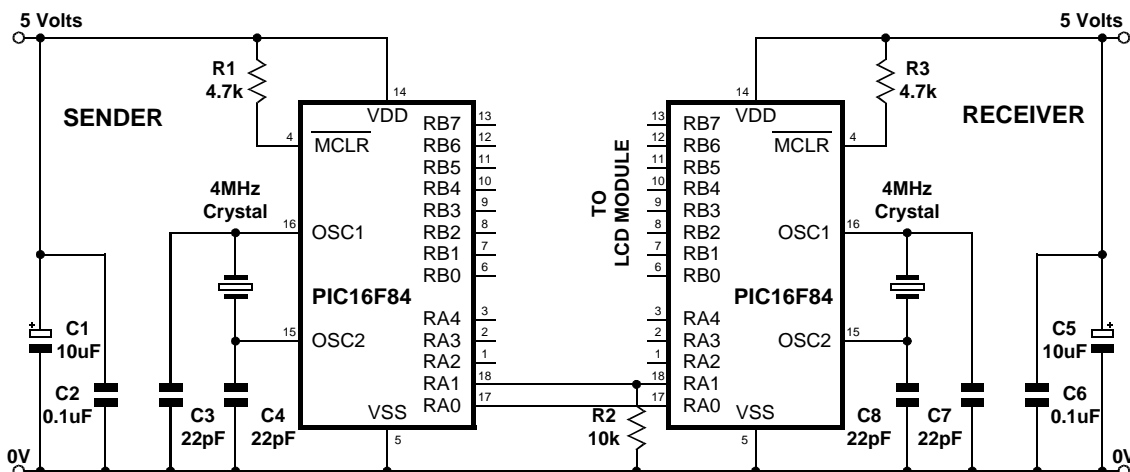
```

When **Serin** executes, bit-0 of PORTA (*Rpin*) is made an input in preparation for incoming data, and bit-1 of PORTA (*Fpin*) is made an output low, to signal “go” to the sender. After **Serin** finishes receiving data, bit-1 of PORTA is brought high to notify the sender to stop. If an inverted *BaudMode* had been specified, the *Fpin*’s responses would have been reversed. The table below illustrates the relationship of serial polarity to *Fpin* states.

Serial Polarity	Ready to Receive ("Go")	Not Ready to Receive ("Stop")
Inverted	<i>Fpin</i> is High (1)	<i>Fpin</i> is Low (0)
Non-inverted	<i>Fpin</i> is Low (0)	<i>Fpin</i> is High (1)

See the following circuit for a flow control example using two 16F84 devices. In the demonstration program example, the sender transmits the whole word “HELLO!” in approx 6 ms. The receiver catches the first byte at most; by the time it got back from the first 1-second delay (**DelayMs** 1000), the rest of the data would be long gone. With flow control, communication is flawless since the sender waits for the receiver to catch up.

In the circuit below, the flow control pin (PORTA.1) is pulled to ground through a 10k resistor. This is to ensure that the sender sees a stop signal (0 for inverted communications) when the receiver is first powered up.



Communicating Communication between two PICs using flow control.

' *Sender Code. Program into the Sender device.*

```
Loop:
Serout PORTA.0\PORTA.1, 16468, ["HELLO!"] ' Send the message.
DelayMs 2500 ' Delay for 2.5 seconds
Goto Loop ' Repeat the message forever
```

' *Receiver Code. Program into the Receiver device.*

```
Dim Message as Byte
Again:
Serin PORTA.0\PORTA.1, 16468, [Message] ' Get 1 byte.
Print Message ' Display the byte on LCD.
DelayMs 1000 ' Delay for 1 second.
Goto Again ' Repeat forever
```

Serin Modifiers.

The **Serin** command can be configured to wait for a specified sequence of characters before it retrieves any additional input. For example, suppose a device attached to the PICmicro™ is known to send many different sequences of data, but the only data you wish to observe happens to appear right after the unique characters, "XYZ". A modifier named **Wait** can be used for this purpose: -

```
Serin PORTA.0, 16468, [Wait("XYZ"), SerData]
```

The above code waits for the characters "X", "Y" and "Z" to be received, in that order, then it receives the next data byte and places it into variable SerData.

The compiler also has a modifier for handling a string of characters, named **Str**.

The **Str** modifier is used for receiving a string of characters into a byte array variable.

A string is a set of characters that are arranged or accessed in a certain order. The characters "ABC" would be stored in a string with the "A" first, followed by the "B" then followed by the "C". A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string "ABC" would be stored in a byte array containing three bytes (elements).

Below is an example that receives ten bytes through bit-0 of PORTA at 9600 bps, N81/inverted, and stores them in the 10-byte array, SERString: -

```
Dim SerString[10] as Byte      ' Create a 10-byte array.
Serin PORTA.0, 16468, [Str SerString] ' Fill the array with data.
Print Str SerString           ' Display the string.
```

If the amount of received characters is not enough to fill the entire array, then a formatter may be placed after the array's name, which will only receive characters until the specified length is reached. For example: -

```
Dim SerString[10] as Byte      ' Create a 10-byte array.
Serin PORTA.0, 16468, [Str SerString\5] ' Fill first 5-bytes of array
Print Str SerString\5         ' Display the 5-character string.
```

The example above illustrates how to fill only the first n bytes of an array, and then how to display only the first n bytes of the array. n refers to the value placed after the backslash.

Because of its complexity, serial communication can be rather difficult to work with at times. Using the guidelines below when developing a project using the **Serin** and **Serout** commands may help to eliminate some obvious errors: -

Always build your project in steps.

Start with small, manageable pieces of code, (that deal with serial communication) and test them, one individually.

Add more and more small pieces, testing them each time, as you go.

Never write a large portion of code that works with serial communication without testing its smallest workable pieces first.

Pay attention to timing.

Be careful to calculate and overestimate the amount of time, operations should take within the microcontroller for a given oscillator frequency. Misunderstanding the timing constraints is the source of most problems with code that communicate serially. If the serial communication in your project is bi-directional, the above statement is even more critical.

Pay attention to wiring.

Take extra time to study and verify serial communication wiring diagrams. A mistake in wiring can cause strange problems in communication, or no communication at all. Make sure to connect the ground pins (Vss) between the devices that are communicating serially.

Verify port setting on the PC and in the Serin / Serout commands.

Unmatched settings on the sender and receiver side will cause garbled data transfers or no data transfers. This is never more critical than when a line transceiver is used (i.e. MAX232). Always remember that a line transceiver inverts the serial polarity.

If the serial data received is unreadable, it is most likely caused by a baud rate setting error, or a polarity error. If receiving data from another device that is not a PICmicro™, try to use baud rates of 9600 and below, or alternatively, use a higher frequency crystal.

Because of additional overheads in the microcontroller, and the fact that the **Serin** command offers no hardware receive buffer for serial communication, received data may sometimes be missed or garbled. If this occurs, try lowering the baud rate, or increasing the crystal frequency. Using simple variables (not arrays) will also increase the chance that the device will receive the data properly.

See also : Hrsin, Hrsout, Hserin, Hserout, Rsin, Rsout.

Serout

Syntax

Serout *Tpin* { \ *Fpin* }, *Baudmode*, { *Pace*, } { *Timeout*, *Tlabel*, } [*OutputData*]

Overview

Transmit asynchronous serial data (i.e. RS232 data).

Operators

Tpin is a Port.Bit constant that specifies the I/O pin through which the serial data will be transmitted. This pin will be set to output mode while operating. The state of this pin when finished is determined by the driver bit in *Baudmode*.

Fpin is an optional Port.Bit constant that specifies the I/O pin to monitor for flow control status. This pin will be set to input mode. Note: *Fpin* must be specified in order to use the optional *Timeout* and *Tlabel* operators in the **Serout** command.

Baudmode may be a variable, constant, or expression (0 - 65535) that specifies serial timing and configuration.

Pace is an optional variable, constant, or expression (0 - 65535) that determines the length of the delay between transmitted bytes. Note: *Pace* cannot be used simultaneously with *Timeout*.

Timeout is an optional variable or constant (0 - 65535) that informs **Serout** how long to wait for *Fpin* permission to send. If permission does not arrive in time, the program will jump to the address specified by *Tlabel*. Note: *Fpin* must be specified in order to use the optional *Timeout* and *Tlabel* operators in the **Serout** command.

Tlabel is an optional label that must be provided along with *Timeout*. *Tlabel* indicates where the program should jump to in the event that permission to send data is not granted within the period specified by *Timeout*.

OutputData is list of variables, constants, expressions and modifiers that informs **Serout** how to format outgoing data. **Serout** can transmit individual or repeating bytes, convert values into decimal, hex or binary text representations, or transmit strings of bytes from variable arrays, and **Cdata** constructs. These actions can be combined in any order in the *OutputData* list.

Notes

One of the most popular forms of communication between electronic devices is serial communication. There are two major types of serial communication; asynchronous and synchronous. The **Rsin**, **Rsout**, **Serin** and **Serout** commands are all used to send and receive asynchronous serial data. While the **Shin** and **Shout** commands are for use with synchronous communications.

The term asynchronous means 'no clock.' More specifically, 'asynchronous serial communication' means data is transmitted and received without the use of a separate 'clock' line. Data can be sent using as few as two wires; one for data and one for ground. The PC's serial ports (also called COM ports or RS232 ports) use asynchronous serial communication. Note: the other kind of serial communication, synchronous, uses at least three wires; one for clock, one for data and one for ground.

RS232 is the electrical specification for the signals that PC serial ports use. Unlike standard TTL logic, where 5 volts is a logic 1 and 0 volts is logic 0, RS232 uses -12 volts for logic 1 and +12 volts for logic 0. This specification allows communication over longer wire lengths without amplification.

Most circuits that work with RS232 use a line driver / receiver (transceiver). This component does two things: -

- Convert the ± 12 volts of RS-232 to TTL compatible 0 to 5 volt levels.
- Invert the voltage levels, so that 5 volts = logic 1 and 0 volts = logic 0.

By far, the most common line driver device is the MAX232 from MAXIM semiconductor. With the addition of a few capacitors, a complete 2-way level converter is realised (see **Serin** for circuit).

The MAX232 is not the only device available, there are other types that do not require any external capacitors at all. Visit Maxim's excellent web site at www.maxim.com <<http://www.maxim.com>>, and download one of their many detailed datasheets.

Because of the excellent IO capabilities of the PICmicro™ range of devices, and the adoption of TTL levels on most modern PC serial ports, a line driver is often unnecessary unless long distances are involved between the transmitter and the receiver. Instead a simple current limiting resistor is all that's required (see **Serin** for circuit).

You should remember that when using a line transceiver such as the MAX232, the serial mode (polarity) is inverted in the process of converting the signal levels, however, if using the direct connection, the mode is untouched. This is the single most common cause of errors when connecting serial devices, therefore you must make allowances for this within your software.

Asynchronous serial communication relies on precise timing. Both the sender and receiver must be set for identical timing, this is commonly expressed in bits per second (bps) called baud. **Serout** requires a value called *Baudmode* that informs it of the relevant characteristics of the incoming serial data; the bit period, number of data and parity bits, and polarity.

The *Baudmode* argument for **Serout** accepts a 16-bit value that determines its characteristics: 1-stop bit, 8-data bits/no-parity or 7-data bits/even-parity and virtually any speed from as low as 300 baud to 38400 baud (depending on the crystal frequency used). Table 2 below shows how *Baudmode* is calculated, while table 3 shows some common *baudmodes* for standard serial baud rates.

Step 1.	Determine the bit period. (bits 0 – 11)	$(1,000,000 / \text{baud rate}) - 20$
Step 2.	data bits and parity. (bit 13)	8-bit/no-parity = step 1 + 0 7-bit/even-parity = step 1 + 8192
Step 3.	Select polarity. (bit 14)	True (noninverted) = step 2 + 0 Inverted = step 2 + 16384

Baudmode calculation.

Add the results of steps 1, 2, and 3 to determine the correct value for the *Baudmode* operator

BaudRate	8-bit no-parity inverted	8-bit no-parity true	7-bit even-parity inverted	7-bit even-parity true
300	19697	3313	27889	11505
600	18030	1646	26222	9838
1200	17197	813	25389	9005
2400	16780	396	24972	8588
4800	16572	188	24764	8380
9600	16468	84	24660	8276

Note

For 'open' baudmodes used in networking, add 32768 to the values from the previous table.

If communications are with existing software or hardware, its speed and mode will determine the choice of baud rate and mode. In general, 7-bit/even-parity (7E) mode is used for text, and 8-bit/no-parity (8N) for byte-oriented data. Note: the most common mode is 8-bit/no-parity, even when the data transmitted is just text. Most devices that use a 7-bit data mode do so in order to take advantage of the parity feature. Parity can detect some communication errors, but to use it you lose one data bit. This means that incoming data bytes transferred in 7E (even-parity) mode can only represent values from 0 to 127, rather than the 0 to 255 of 8N (no-parity) mode.

The compiler's serial commands **Serout** and **Serin**, have the option of still using a parity bit with 4 to 8 data bits. This is through the use of a **Declare**: -

With parity disabled (the default setting): -

```
Declare Serial_Data 4 ' Set Serout and Serin data bits to 4
Declare Serial_Data 5 ' Set Serout and Serin data bits to 5
Declare Serial_Data 6 ' Set Serout and Serin data bits to 6
Declare Serial_Data 7 ' Set Serout and Serin data bits to 7
Declare Serial_Data 8 ' Set Serout and Serin data bits to 8 (default)
```

With parity enabled: -

```
Declare Serial_Data 5 ' Set Serout and Serin data bits to 4
Declare Serial_Data 6 ' Set Serout and Serin data bits to 5
Declare Serial_Data 7 ' Set Serout and Serin data bits to 6
Declare Serial_Data 8 ' Set Serout and Serin data bits to 7 (default)
Declare Serial_Data 9 ' Set Serout and Serin data bits to 8
```

Serial_Data data bits may range from 4 bits to 8 (the default if no **Declare** is issued). Enabling parity uses one of the number of bits specified.

Declaring **Serial_Data** as 9 allows 8 bits to be read and written along with a 9th parity bit.

Parity is a simple error-checking feature. When the **Serout** command's *Baudmode* is set for even parity (compiler default) it counts the number of 1s in the outgoing byte and uses the parity bit to make that number even. For example, if it is sending the 7-bit value: %0011010, it sets the parity bit to 1 in order to make an even number of 1s (four).

The receiver also counts the data bits to calculate what the parity bit should be. If it matches the parity bit received, the serial receiver assumes that the data was received correctly. Of course, this is not necessarily true, since two incorrectly received bits could make parity seem correct when the data was wrong, or the parity bit itself could be bad when the rest of the data was correct. Parity errors are only detected on the receiver side.

Normally, the receiver determines how to handle an error. In a more robust application, the receiver and transmitter might be set up in such that the receiver can request a re-send of data that was received with a parity error.

Serout Modifiers.

The example below will transmit a single byte through bit-0 of PORTA at 2400 baud, 8N1, inverted: -

```
Serout PORTA.0, 16780, [65]
```

In the above example, **Serout** will transmit a byte equal to 65 (the ASCII value of the character "A") through PORTA.0. If the PICmicro™ was connected to a PC running a terminal program such as HyperTerminal set to the same baud rate, the character "A" would appear on the screen. Always remembering that the polarity will differ if a line transceiver such as the MAX232 is used.

What if you wanted the value 65 to appear on the PC's screen? As was stated earlier, it is up to the receiving side (in serial communication) to interpret the values. In this case, the PC is interpreting the byte-sized value to be the ASCII code for the character "A". Unless you're also writing the software for the PC, you cannot change how the PC interprets the incoming serial data, therefore to solve this problem, the data needs to be translated before it is sent.

The **Serout** command provides a modifier which will translate the value 65 into two ASCII codes for the characters "6" and "5" and then transmit them: -

```
Serout PORTA.0, 16780, [@ 65]
```

or

```
Serout PORTA.0, 16780, [Dec 65]
```

Notice that the decimal modifier in the **Serout** command is the character @ or word **Dec**, both these modifiers do the same thing, which is to inform **Serout** to convert the number into separate ASCII characters which represent the value in decimal form. If the value 65 in the code were changed to 123, the **Serout** command would send three bytes (49, 50 and 51) corresponding to the characters "1", "2" and "3".

This is exactly the same modifier that is used in the **Rsout** and **Print** commands.

As well as the **Dec** modifier, **Serout** may use **Hex**, or **Bin** modifiers, again, these are the same as used in the **Rsout** and **Print** commands. Therefore, please refer to the **Rsout** or **Print** command descriptions for an explanation of these. The **Serout** command sends quoted text exactly as it appears in the *OutputData* list:

```
Serout PORTA.0, 16780, ["HELLO WORLD", 13]  
Serout PORTA.0, 16780, ["Num = ", Dec 100]
```

The above code will display "HELLO WORLD" on one line and "Num = 100" on the next line. Notice that you can combine data to output in one **Serout** command, separated by commas. In the example above, we could have written it as one line of code: -

```
Serout PORTA.0, 16780, ["HELLO WORLD", 13, "Num = ", Dec 100]
```

Serout also has some other modifiers. These are listed below: -

Modifier	Operation
At ypos,xpos	Position the cursor on a serial LCD
Cls	Clear a serial LCD (also creates a 30ms delay)
Bin{1..32}	Send binary digits
Dec{1..10}	Send decimal digits
Hex{1..8}	Send hexadecimal digits
Sbin{1..32}	Send signed binary digits
Sdec{1..10}	Send signed decimal digits
Shex{1..8}	Send signed hexadecimal digits
Ibin{1..32}	Send binary digits with a preceding '%' identifier
Idec{1..10}	Send decimal digits with a preceding '#' identifier
Ihex{1..8}	Send hexadecimal digits with a preceding '\$' identifier
ISbin{1..32}	Send signed binary digits with a preceding '%' identifier
ISdec{1..10}	Send signed decimal digits with a preceding '#' identifier
IShex{1..8}	Send signed hexadecimal digits with a preceding '\$' identifier
Rep c\n	Send character c repeated n times

If a floating point variable is to be displayed, then the digits after the **Dec** modifier determine how many remainder digits are printed. i.e. numbers after the decimal point.

```
Dim Flt as Float
Flt = 3.145
Serout PORTA.0, 16780, [Dec2 Flt] ' Send 2 values after the decimal point
```

The above program will send 3.14

If the digit after the **Dec** modifier is omitted, then 3 values will be displayed after the decimal point.

```
Dim Flt as Float
Flt = 3.1456
Serout PORTA.0, 16780, [Dec Flt] ' Send 3 values after the decimal point
```

The above program will send 3.145

There is no need to use the **Sdec** modifier for signed floating point values, as the compiler's **Dec** modifier will automatically display a minus result: -

```
Dim Flt as Float
Flt = -3.1456
Serout PORTA.0, 16780, [Dec Flt] ' Send 3 values after the decimal point
```

The above program will send -3.145

Hex or **Bin** modifiers cannot be used with floating point values or variables.

Using Strings with Serout.

The **Str** modifier is used for transmitting a string of characters from a byte array variable. A string is a set of characters that are arranged or accessed in a certain order. The characters "ABC" would be stored in a string with the "A" first, followed by the "B" then followed by the "C". A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string "ABC" would be stored in a byte array containing three bytes (elements).

Below is an example that transmits five bytes (from a byte array) through bit-0 of PORTA at 9600 bps, N81/inverted: -

```
Dim SerString[10] as Byte      ' Create a 10-byte array.
SerString[0] = "H"            ' Load the first 5 bytes of the array
SerString[1] = "E"            ' With the word "HELLO"
SerString[2] = "L"
SerString[3] = "L"
SerString[4] = "O"
Serout PORTA.0, 16468, [Str SerString\5] ' Send 5-byte string.
```

Note that we use the optional `\n` argument of **Str**. If we didn't specify this, the PICmicro™ would try to keep sending characters until all 10 bytes of the array were transmitted, or it found a byte equal to 0 (a null terminator). Since we didn't specify a last byte of 0 in the array, and we do not wish the last five bytes to be transmitted, we chose to tell it explicitly to only send the first 5 characters.

The above example may also be written as: -

```
Dim SerString[10] as Byte      ' Create a 10-byte array.
Str SerString = "HELLO", 0     ' Load the first 6 bytes of the array
Serout PORTA.0, 16468, [Str SerString] ' Send first 5-bytes of string.
```

In the above example, we specifically added a null terminator to the end of the string (a zero). Therefore, the **Str** modifier within the **Serout** command will output data until this is reached. An alternative to this would be to create the array exactly the size of the text. In our example, the array would have been 5 elements in length.

Another form of string is used by the **Cstr** modifier. Note: Because this uses the **Cdata** command to create the individual elements it is only for use with PICs that support self-modifying features, such as the 16F87X, and 18XXXX range of devices.

Below is an example of using the **Cstr** modifier. It's function is the same as the above examples, however, no RAM is used for creating arrays.

```
Serout PORTA.0, 16468, [Cstr SerString]

SerString: Cdata "HELLO", 0
```

The **Cstr** modifier will always be terminated by a null (i.e. zero at the end of the text or data). If the null is omitted, then the **Serout** command will continue transmitting characters forever.

The **Serout** command can also be configured to pause between transmitted bytes. This is the purpose of the optional *Pace* operator. For example (9600 baud N8, inverted): -


```
Serout PORTA.0, 16468, 1000, ["Send this message Slowly"]
```

Here, the PICmicro™ transmits the message "Send this message Slowly" with a 1 second delay between each character.

A good reason to use the *Pace* feature is to support devices that require more than one stop bit. Normally, the PICmicro™ sends data as fast as it can (with a minimum of 1 stop bit between bytes). Since a stop bit is really just a resting state in the line (no data transmitted), using the *Pace* option will effectively add multiple stop bits. Since the requirement for 2 or more stop bits (on some devices) is really just a minimum requirement, the receiving side should receive this data correctly.

Serout Flow Control.

When designing an application that requires serial communication between PICs, you need to work within these limitations: -

When the PICmicro™ is sending or receiving data, it cannot execute other instructions. When the PICmicro™ is executing other instructions, it cannot send or receive data. The compiler does not offer a serial buffer as there is in PCs. At lower crystal frequencies, and higher serial rates, the PICmicro™ cannot receive data via **Serin**, process it, and execute another **Serin** in time to catch the next chunk of data, unless there are significant pauses between data transmissions.

These limitations can sometimes be addressed by using flow control; the *Fpin* option for **Serout** and **Serin**. Through *Fpin*, **Serin** can inform another PICmicro™ sender when it is ready to receive data and **Serout** (on the sender) will wait for permission to send. *Fpin* flow control follows the rules of other serial handshaking schemes, however most computers other than the PICmicro™ cannot start and stop serial transmission on a byte-by-byte basis. That is why this discussion is limited to communication between PICmicros.

Below is an example using flow control with data through bit-0 of PORTA, and flow control through bit-1 of PORTA, 9600 baud, N8, noninverted: -

```
Serout PORTA.0\PORTA.1, 84, [SerData]
```

When **Serin** executes, bit-0 of PORTA (*Tpin*) is made an output in preparation for sending data, and bit-1 of PORTA (*Fpin*) is made an input, to wait for the "go" signal from the receiver. The table below illustrates the relationship of serial polarity to *Fpin* states.

Serial Polarity	Ready to Receive ("Go")	Not Ready to Receive ("Stop")
Inverted	<i>Fpin</i> is High (1)	<i>Fpin</i> is Low (0)
Non-inverted	<i>Fpin</i> is Low (0)	<i>Fpin</i> is High (1)

See **Serin** for a flow control circuit.

The **Serout** command supports open-drain and open-source output, which makes it possible to network multiple PICs on a single pair of wires. These 'open baudmodes' only actively drive the *Tpin* in one state (for the other state, they simply disconnect the pin; setting it to an input mode). If two PICs in a network had their **Serout** lines connected together (while a third device listened on that line) and the PICs were using always-driven baudmodes, they could simultaneously output two opposite states (i.e. +5 volts and ground). This would create a short circuit. The heavy current flow would likely damage the I/O pins or the PICs themselves.

Since the open baudmodes only drive in one state and float in the other, there's no chance of this kind of short happening.

The polarity selected for **Serout** determines which state is driven and which is open as shown in the table below.

Serial Polarity	State(0)	State(1)	Resistor Pulled to:
Inverted	Open	Driven	Gnd (Vss)
Non-inverted	Driven	Open	+5V (Vdd)

Since open baudmodes only drive to one state, they need a resistor to pull the networked line into the opposite state, as shown in the above table and in the circuits below. Open baudmodes allow the PICmicro™ to share a line, however it is up to your program to resolve other networking issues such as who talks when, and how to detect, prevent and fix data errors.

See also : **Rsin, Rsout, Hrsin, Hrsout, Hserin, Hserout, Serin.**

Servo

Syntax

Servo *Pin, Rotation Value*

Overview

Control a remote control type servo motor.

Operators

Pin is a Port.Pin constant that specifies the I/O pin for the attachment of the motor's control terminal.

Rotation Value is a 16-bit (0-65535) constant or **Word** variable that dictates the position of the motor. A value of approx 500 being a rotation to the farthest position in a direction and approx 2500 being the farthest rotation in the opposite direction. A value of 1500 would normally centre the servo but this depends on the motor type.

Example

```
' Control a servo motor attached to pin 3 of PORTA

Device = 16F628           ' We'll use a 14-bit core device
Dim Pos as Word         ' Servo Position
Symbol Pin = PORTA.3    ' Alias the servo pin
CMCON = 7               ' PORTA to digital
Cls                     ' Clear the LCD
Pos = 1500              ' Centre the servo
PORTA = 0               ' PORTA lines low to read buttons
TrisA = %00000111      ' Enable the button pins as inputs

' Check any button pressed to move servo
Main:
If PORTA.0 = 0 And Pos < 3000 Then Pos = Pos + 1 ' Move servo left
If PORTA.1 = 0 Then Pos = 1500                  ' Centre servo
If PORTA.2 = 0 And Pos > 0 Then Pos = Pos - 1   ' Move servo right
Servo Pin, Pos
DelayMs 5                                       ' Servo update rate
Print At 1, 1, "Position=", Dec Pos, " "
Goto Main
```

Notes

Servos of the sort used in radio-controlled models are finding increasing applications in this robotics age we live in. They simplify the job of moving objects in the real world by eliminating much of the mechanical design. For a given signal input, you get a predictable amount of motion as an output.

To enable a servo to move it must be connected to a 5 Volt power supply capable of delivering an ampere or more of peak current. It then needs to be supplied with a positioning signal. The signal is normally a 5 Volt, positive-going pulse between 1 and 2 milliseconds (ms) long, repeated approximately 50 times per second.

The width of the pulse determines the position of the servo. Since a servo's travel can vary from model to model, there is not a definite correspondence between a given pulse width and a particular servo angle, however most servos will move to the centre of their travel when receiving 1.5ms pulses.

Servos are closed-loop devices. This means that they are constantly comparing their commanded position (proportional to the pulse width) to their actual position (proportional to the resistance of an internal potentiometer mechanically linked to the shaft). If there is more than a small difference between the two, the servo's electronics will turn on the motor to eliminate the error. In addition to moving in response to changing input signals, this active error correction means that servos will resist mechanical forces that try to move them away from a commanded position. When the servo is unpowered or not receiving positioning pulses, the output shaft may be easily turned by hand. However, when the servo is powered and receiving signals, it won't move from its position.

Driving servos with Proton is extremely easy. The **Servo** command generates a pulse in 1microsecond (μ s) units, so the following code would command a servo to its centred position and hold it there: -

Again:

```
Servo PORTA.0, 1500  
DelayMs 20  
Goto Again
```

The 20ms delay ensures that the program sends the pulse at the standard 50 pulse-per-second rate. However, this may be lengthened or shortened depending on individual motor characteristics.

The **Servo** command is oscillator independent and will always produce 1us pulses regardless of the crystal frequency used.

See also : [Pulseout](#).

SetBit

Syntax

SetBit *Variable, Index*

Overview

Set a bit of a variable or register using a variable index to the bit of interest.

Operators

Variable is a user defined variable, of type **Byte**, **Word**, or **Dword**.

Index is a constant, variable, or expression that points to the bit within *Variable* that requires setting.

Example

```
' Clear then Set each bit of variable ExVar
Device = 16F877
Declare Xtal = 4
Dim ExVar as Byte
Dim Index as Byte
Cls
ExVar = %11111111
Again:
For Index = 0 to 7           ' Create a loop for 8 bits
    ClearBit ExVar, Index   ' Clear each bit of ExVar
    Print At 1,1,Bin8 ExVar ' Display the binary result
    DelayMs 100            ' Slow things down to see what's happening
Next                         ' Close the loop
For Index = 7 to 0 Step -1 ' Create a loop for 8 bits
    SetBit ExVar, Index     ' Set each bit of ExVar
    Print At 1,1,Bin8 ExVar ' Display the binary result
    DelayMs 100            ' Slow things down to see what's happening
Next                         ' Close the loop
Goto Again                  ' Do it forever
```

Notes

There are many ways to set a bit within a variable, however, each method requires a certain amount of manipulation, either with rotates, or alternatively, the use of indirect addressing using the FSR, and INDF registers. Each method has its merits, but requires a certain amount of knowledge to accomplish the task correctly. The **SetBit** command makes this task extremely simple using a register rotate method, however, this is not necessarily the quickest method, or the smallest, but it is the easiest. For speed and size optimisation, there is no shortcut to experience.

To Set a known constant bit of a variable or register, then access the bit directly using Port.n.

```
PORTA.1 = 1
```

or

```
Var1.4 = 1
```

If a Port is targeted by **SetBit**, the Tris register is **not** affected.

See also : **ClearBit, GetBit, LoadBit.**

Set_OSCCAL

Syntax

Set_OSCCAL

Overview

Calibrate the on-chip oscillator found on some PICmicro™ devices.

Notes

Some devices, such as the PIC12C67x or 16F62x range, have on-chip RC oscillators. These devices contain an oscillator calibration factor in the last location of code space. The on-chip oscillator may be fine-tuned by reading the data from this location and moving it into the OSCCAL register. The command **Set_OSCCAL** has been specially created to perform this task automatically each time the program starts: -

```
Device = 12C671
Set_OSCCAL      ' Set OSCCAL for 1K device 12C671
```

Add this command near the beginning of the program to perform the setting of OSCCAL.

If a UV erasable (windowed) device has been erased, the value cannot be read from memory. To set the OSCCAL register on an erased part, add the following line near the beginning of the program: -

```
OSCCAL = $C0      ' Set OSCCAL register to $C0
```

The value \$C0 is only an example. The part would need to be read before it is erased to obtain the actual OSCCAL value for that particular device.

Always refer to the device's data sheet for more information on OSCCAL.

Set

Syntax

Set *Variable* or *Variable.Bit*

Overview

Place a variable or bit in a high state. For a variable, this means setting all the bits to 1. For a bit this means setting it to 1.

Operators

Variable can be any variable or register.

Variable.Bit can be any variable and bit combination.

Example

```
Set Var1.3      ' Set bit 3 of Var1
Set Var1        ' Load Var1 with the value of 255
Set STATUS.0    ' Set the carry flag high
Set Array       ' Set all of an Array variable. i.e. set to 255 or 65535
Set String1     ' Set all of a String variable. i.e. set to spaces (ASCII 32)
Set             ' Load all RAM with 255
```

Notes

Set does not alter the Tris register if a Port is targeted.

If no variable follows the **Set** command then all user RAM will be loaded with the value 255.

See also : Clear, High, Low.

Shin

Syntax

Shin *dpin, cpin, mode, [result { \bits } { ,result { \bits }...}]*

or

Var = **Shin** *dpin, cpin, mode, shifts*

Overview

Shift data in from a synchronous-serial device.

Operators

Dpin is a Port.Pin constant that specifies the I/O pin that will be connected to the synchronous-serial device's data output. This pin's I/O direction will be changed to input and will remain in that state after the instruction is completed.

Cpin is a Port.Pin constant that specifies the I/O pin that will be connected to the synchronous-serial device's clock input. This pin's I/O direction will be changed to output.

Mode is a constant that tells **Shin** the order in which data bits are to be arranged and the relationship of clock pulses to valid data. Below are the symbols, values, and their meanings: -

Symbol	Value	Description
MsbPre MsbPre_L	0	Shift data in highest bit first. Read data before sending clock. Clock idles low
LsbPre LsbPre_L	1	Shift data in lowest bit first. Read data before sending clock. Clock idles low
MsbPost MsbPost_L	2	Shift data in highest bit first. Read data after sending clock. Clock idles low
LsbPost LsbPost_L	3	Shift data in highest bit first. Read data after sending clock. Clock idles low
MsbPre_H	4	Shift data in highest bit first. Read data before sending clock. Clock idles high
LsbPre_H	5	Shift data in lowest bit first. Read data before sending clock. Clock idles high
MsbPost_H	6	Shift data in highest bit first. Read data after sending clock. Clock idles high
LsbPost_H	7	Shift data in lowest bit first. Read data after sending clock. Clock idles high

Result is a bit, byte, or word variable in which incoming data bits will be stored.

Bits is an optional constant specifying how many bits (1-16) are to be input by **Shin**. If no *bits* entry is given, **Shin** defaults to 8 bits.

Shifts informs the **Shin** command as to how many bit to shift in to the assignment variable, when used in the inline format.

Notes

Shin provides a method of acquiring data from synchronous-serial devices, without resorting to the hardware SPI modules resident on some PICmicro™ types. Data bits may be valid after the rising or falling edge of the clock line. This kind of serial protocol is commonly used by controller peripherals such as ADCs, DACs, clocks, memory devices, etc.

The **Shin** instruction causes the following sequence of events to occur: -

Makes the clock pin (cpin) output low.

Makes the data pin (dpin) an input.

Copies the state of the data bit into the msb (lsb-modes) or lsb (msb modes) either before (-pre modes) or after (-post modes) the clock pulse.

Pulses the clock pin high.

Shifts the bits of the result left (msb- modes) or right (lsb-modes).

Repeats the appropriate sequence of getting data bits, pulsing the clock pin, and shifting the result until the specified number of bits is shifted into the variable.

Making **Shin** work with a particular device is a matter of matching the mode and number of bits to that device's protocol. Most manufacturers use a timing diagram to illustrate the relationship of clock and data.

```
Symbol CLK = PORTB.0
Symbol DTA = PORTB.1
Shin DTA, CLK, MsbPre, [Var1] ' Shift in msb-first, pre-clock.
```

In the above example, both **Shin** instructions are set up for msb-first operation, so the first bit they acquire ends up in the msb (leftmost bit) of the variable.

The post-clock Shift in, acquires its bits after each clock pulse. The initial pulse changes the data line from 1 to 0, so the post-clock Shiftin returns %01010101.

By default, **Shin** acquires eight bits, but you can set it to shift any number of bits from 1 to 16 with an optional entry following the variable name. In the example above, substitute this for the first **Shin** instruction: -

```
Shin DTA, CLK, MsbPre, [Var1\4] ' Shift in 4 bits.
```

Some devices return more than 16 bits. For example, most 8-bit shift registers can be daisy-chained together to form any multiple of 8 bits; 16, 24, 32, 40... You can use a single **Shin** instruction with multiple variables.

Each variable can be assigned a particular number of bits with the backslash (\) option. Modify the previous example: -

```
' 5 bits into Var1; 8 bits into Var2.
Shin DTA, CLK, MsbPre, [Var1\5, Var2]
Print "1st variable: ", Bin8 Var1
Print "2nd variable: ", Bin8 Var2
```

Inline Shin Command.

The structure of the inline **Shin** command is: -

Var = **Shin** dpin, cpin, mode, shifts

DPin, *CPin*, and *Mode* have not changed in any way, however, the **INLINE** structure has a new operand, namely *Shifts*. This informs the **Shin** command as to how many bit to shift in to the assignment variable. For example, to shift in an 8-bit value from a serial device, we would use: -

```
Var1 = Shin DTA, CLK, MsbPre, 8
```

To shift 16-bits into a **Word** variable: -

```
Wrd = Shin DTA, CLK, MsbPre, 16
```

Shout

Syntax

Shout *Dpin*, *Cpin*, *Mode*, [*OutputData* {\Bits} {, *OutputData* {\Bits}..}]

Overview

Shift data out to a synchronous serial device.

Operators

Dpin is a Port.Pin constant that specifies the I/O pin that will be connected to the synchronous serial device's data input. This pin will be set to output mode.

Cpin is a Port.Pin constant that specifies the I/O pin that will be connected to the synchronous serial device's clock input. This pin will be set to output mode.

Mode is a constant that tells **Shout** the order in which data bits are to be arranged. Below are the symbols, values, and their meanings: -

Symbol	Value	Description
LsbFirst LsbFirst_L	0	Shift data out lowest bit first. Clock idles low
MsbFirst MsbFirst_L	1	Shift data out highest bit first. Clock idles low
LsbFirst_H	4	Shift data out lowest bit first. Clock idles high
MsbFirst_H	5	Shift data out highest bit first. Clock idles high

OutputData is a variable, constant, or expression containing the data to be sent.

Bits is an optional constant specifying how many bits are to be output by **Shout**. If no *Bits* entry is given, **Shout** defaults to 8 bits.

Notes

Shin and **Shout** provide a method of acquiring data from synchronous serial devices. Data bits may be valid after the rising or falling edge of the clock line. This kind of serial protocol is commonly used by controller peripherals like ADCs, DACs, clocks, memory devices, etc.

At their heart, synchronous-serial devices are essentially shift-registers; trains of flip flops that receive data bits in a bucket brigade fashion from a single data input pin. Another bit is input each time the appropriate edge (rising or falling, depending on the device) appears on the clock line.

The **Shout** instruction first causes the clock pin to output low and the data pin to switch to output mode. Then, **Shout** sets the data pin to the next bit state to be output and generates a clock pulse. **Shout** continues to generate clock pulses and places the next data bit on the data pin for as many data bits as are required for transmission.

Making **Shout** work with a particular device is a matter of matching the mode and number of bits to that device's protocol. Most manufacturers use a timing diagram to illustrate the relationship of clock and data. One of the most important items to look for is which bit of the data should be transmitted first; most significant bit (MSB) or least significant bit (LSB).

Example

```
Shout DTA, CLK, MsbFirst, [250]
```

In the above example, the **Shout** command will write to I/O pin DTA (the *Dpin*) and will generate a clock signal on I/O CLK (the *Cpin*). The **Shout** command will generate eight clock pulses while writing each bit (of the 8-bit value 250) onto the data pin (*Dpin*). In this case, it will start with the most significant bit first as indicated by the *Mode* value of **MsbFirst**.

By default, **Shout** transmits eight bits, but you can set it to shift any number of bits from 1 to 16 with the *Bits* argument. For example: -

```
Shout DTA, CLK, MsbFirst, [250\4]
```

Will only output the lowest 4 bits (%0000 in this case). Some devices require more than 16 bits. To solve this, you can use a single **Shout** command with multiple values. Each value can be assigned a particular number of bits with the *Bits* argument. As in: -

```
Shout DTA, CLK, MsbFirst, [250\4, 1045\16]
```

The above code will first shift out four bits of the number 250 (%1111) and then 16 bits of the number 1045 (%0000010000010101). The two values together make up a 20 bit value.

See also : **Shin.**

Snooze

Syntax

Snooze *Period*

Overview

Enter sleep mode for a short period. Power consumption is reduced to a few μA assuming no loads are being driven.

Operators

Period is a variable or constant that determines the duration of the reduced power nap. The duration is $(2^{\text{period}}) * 18 \text{ ms}$. (Read as "2 raised to the power of 'period', times 18 ms.") Period can range from 0 to 7, resulting in the following snooze lengths: -

Period	Length of Snooze (approx)
0 - 1	18ms
1 - 2	36ms
2 - 4	72ms
3 - 8	144ms
4 - 16	288ms
5 - 32	576ms
6 - 64	1152ms (1.152 seconds)
7 - 128	2304ms (2.304 seconds)

Example

```
snooze 6 ' Low power mode for approx 1.152 seconds
```

Notes

Snooze intervals are directly controlled by the watchdog timer without compensation. Variations in temperature, supply voltage, and manufacturing tolerance of the device you are using can cause the actual timing to vary by as much as -50% to +100%

See also : **Sleep.**

Sleep

Syntax

Sleep { *Length* }

Overview

Places the microcontroller into low power mode for approx *n* seconds. i.e. power down but leaves the port pins in their previous states.

Operators

Length is an optional variable or constant (1-65535) that specifies the duration of sleep in seconds. If length is omitted, then the Sleep command is assumed to be the assembler mnemonic, which means the microcontroller will sleep continuously, or until the Watchdog timer wakes it up.

Example

```
Symbol LED = PORTA.0
```

Again:

```
High LED      ' Turn LED on.
DelayMs 1000  ' Wait 1 second.
Low LED       ' Turn LED off.
Sleep 60      ' Sleep for 1 minute.
Goto Again
```

Notes

Sleep will place the device into a low power mode for the specified period of seconds. Period is 16 bits, so delays of up to 65,535 seconds are the limit (a little over 18 hours) **Sleep** uses the Watchdog Timer so it is independent of the oscillator frequency. The smallest units is about 2.3 seconds and may vary depending on specific environmental conditions and the device used.

The **Sleep** command is used to put the microcontroller in a low power mode without resetting the registers. Allowing continual program execution upon waking up from the **Sleep** period.

Waking a 14-bit core device from Sleep

All the PICmicro™ range have the ability to be placed into a low power mode, consuming micro Amps of current.

The command for doing this is **Sleep**. The compiler's **Sleep** command or the assembler's **Sleep** instruction may be used. The compiler's **Sleep** command differs somewhat to the assembler's in that the compiler's version will place the device into low power mode for approx *n* seconds (*where n is a value from 0 to 65535*). The assembler's version still places the device into low power mode, however, it does this forever, or until an internal or external source wakes it. This same source also wakes the device when using the compiler's command.

Many things can wake the device from its sleep, the WatchDog Timer is the main cause and is what the compiler's **Sleep** command uses.

Another method of waking the PICmicro™ is an external one, a change on one of the port pins. We will examine more closely the use of an external source. There are several ways of waking the microcontroller using an external source. One is a change on bits 4..7 of PORTB.

Another is a change on bit-0 of PORTB. We shall first look at the wake up on change of PORTB, bits-4..7.

As its name suggests, any change on these pins either high to low or low to high will wake the device. However, to setup this mode of operation several bits within registers INTCON and OPTION_REG need to be manipulated. One of the first things required is to enable the weak PORTB pull-up resistors. This is accomplished by clearing the RBPB bit of OPTION_REG (OPTION_REG.7). If this was not done, then the pins would be floating and random input states would occur waking the microcontroller up prematurely. Although technically we are enabling a form of interrupt, we are not interested in actually running an interrupt handler. Therefore, we must make sure that Global interrupts are disabled, or the device will jump to an interrupt handler every time a change occurs on PORTB. This is done by clearing the GIE bit of INTCON (INTCON.7).

The interrupt we are concerned with is the RB port change type. This is enabled by setting the RBIE bit of the INTCON register (INTCON.3). All this will do is set a flag whenever a change occurs (and of course wake up the PICmicro™). The flag in question is RBIF, which is bit-0 of the INTCON register. For now we are not particularly interested in this flag, however, if global interrupts were enabled, this flag could be examined to see if it was the cause of the interrupt. The RBIF flag is not cleared by hardware so before entering Sleep it should be cleared. It must also be cleared before an interrupt handler is exited.

The **Sleep** command itself is then used. Upon a change of PORTB, bits 4..7 the device will wake up and perform the next instruction (or command) after the **Sleep** command was issued. A second external source for waking the device is a pulse applied to PORTB.0. This interrupt is triggered by the edge of the pulse, high to low or low to high. The INTEDG bit of OPTION_REG (OPTION_REG.6) determines what type of pulse will trigger the interrupt. If it is set, then a low to high pulse will trigger it, and if it is cleared then a high to low pulse will trigger it.

To allow the PORTB.0 interrupt to wake the PICmicro™ the INTE bit must be set, this is bit-4 of the INTCON register. This will allow the flag INTF (INTCON.1) to be set when a pulse with the right edge is sensed. This flag is only of any importance when determining what caused the interrupt. However, it is not cleared by hardware and should be cleared before the Sleep command is used (or the interrupt handler is exited). The program below will wake the microcontroller when a change occurs on PORTB, bits 4-7.

```
Symbol LED = PORTB.0      ' Assign the LED's pin
Symbol RBIF = INTCON.0    ' PORTB[4..7] Change Interrupt Flag
Symbol RBIE = INTCON.3    ' PORTB[4..7] Change Interrupt Enable
Symbol RBPB = OPTION_REG.7 ' PORTB pull-ups
Symbol GIE = INTCON.7     ' Global interrupt enable/disable

Main:
  GIE = 0      ' Turn Off global interrupts
  TRISB.4 = 1  ' Set PORTB.4 as an Input
  RBPB = 0     ' Enable PORTB Pull-up Resistors
  RBIE = 1     ' Enable PORTB[4..7] interrupt

Again:
  DelayMs 100  ' Turn off the LED
  Low LED     ' Clear the PORTB[4..7] interrupt flag
  RBIF = 0    ' Put the PICmicro to sleep
  Sleep      ' When it wakes up, delay for 100ms
  DelayMs 100
  High LED   ' Then light the LED
  Goto Again ' Do it forever
```

SonyIn

Syntax

Variable = **SonyIn**

Overview

Receive Sony SIRC (Sony Infrared Remote Control) data from a predetermined pin. The pin is automatically made an input.

Operators

Variable - a bit, byte, word, dword, or float variable, that will be loaded by **SonyIn**. The return data from the **SonyIn** command consists of two bytes, the System byte containing the type of remote used. i.e. TV, Video etc, and the Command byte containing the actual button value. The order of the bytes is Command (low byte) then System (high byte). If a byte variable is used to receive data from the infrared sensor then only the Command byte will be received.

Example

```
' Receive Sony SIRC data from an infrared sensor attached to PORTC.0
Device = 16F877
Declare SonyIn_Pin = PORTC.0 ' Choose port.pin for infrared sensor
Dim SonyIn_Word as Word ' Create a Word variable to receive the SIRC data
' Alias the Command byte to SonyIn_Word low byte
Dim SonyCommand as SonyIn_Word.Lowbyte
' Alias the System byte to SonyIn_Word high byte
Dim SonySystem as SonyIn_Word.Highbyte

ADCON1 = 7 ' PORTA and PORTE to all digital mode
Cls ' Clear the LCD
While ' Create an infinite loop
  Repeat
    SonyIn_Word = SonyIn ' Receive a signal from the infrared sensor
  Until SonyCommand <> 255 ' Keep looking until a valid header found
  Print at 1,1,"System ",Dec SonySystem," " ' Display the System value
  Print at 2,1,"Command ",Dec SonyCommand," " ' Display the Command value
Wend
```

There is a single Declare for use with **SonyIn**: -

Declare SonyIn_Pin Port . Pin

Assigns the Port and Pin that will be used to input infrared data by the **SonyIn** command. This may be any valid port on the device.

If the Declare is not used in the program, then the default Port and Pin is PORTB.0.

Notes

The **SonyIn** command will return with both Command and System bytes containing 255 if a valid header was not received. The CARRY (STATUS.0) flag will also be set if an invalid header was received. This is an ideal method of determining if the signal received is of the correct type.

SonyIn is oscillator independent as long as the crystal frequency is declared at the top of the program. If no Xtal Declare is used, then **SonyIn** defaults to a 4MHz crystal frequency for its timing.

Sound

Syntax

Sound *Pin*, [*Note,Duration* {,*Note,Duration*...}]

Overview

Generates tone and/or white noise on the specified *Pin*. *Pin* is automatically made an output.

Operators

Pin is a Port.Pin constant that specifies the output pin on the device.

Note can be an 8-bit variable or constant. 0 is silence. *Notes* 1-127 are tones. *Notes* 128-255 are white noise. Tones and white noises are in ascending order (i.e. 1 and 128 are the lowest frequencies, 127 and 255 are the highest). *Note* 1 is approx 78.74Hz and *Note* 127 is approx 10,000Hz.

Duration can be an 8-bit variable or constant that determines how long the *Note* is played in approx 10ms increments.

Example

```
' Star Trek The Next Generation...Theme and ship take-off
Device = 16F877
Declare Xtal = 4

Dim Loop as Byte
Symbol Pin = PORTB.0
```

Theme:

```
Sound Pin, _
[50,60,70,20,85,120,83,40,70,20,50,20,70,20,90,120,90,20,98,160]
DelayMs 500
For Loop = 128 to 255      ' Ascending white noises
    Sound Pin, [Loop,2]    ' For warp drive sound
Next
Sound Pin, [43,80,63,20,77,20,71,80,51,20,_,_
            90,20,85,140,77,20,80,20,85,20,_,_
            90,20,80,20,85,60,90,60,92,60,87,_,_
            60,96,70,0,10,96,10,0,10,96,10,0,_,_
            10,96,30,0,10,92,30,0,10,87,30,0,_,_
            10,96,40,0,20,63,10,0,10,63,10,0,_,_
            10,63,10,0,10,63,20]
DelayMs 10000
Goto Theme
```

Notes

With the excellent I/O characteristics of the PICmicro™, a speaker can be driven through a capacitor directly from the pin of the microcontroller. The value of the capacitor should be determined based on the frequencies of interest and the speaker load. Piezo speakers can be driven directly.

See also : Freqout, DTMFout, Sound2.

Sound2

Syntax

Sound2 Pin2, Pin2, [Note1\Note2\Duration {,Note1,Note2\Duration...}]

Overview

Generate specific notes on each of the two defined pins. With the **Sound2** command more complex notes can be played by the microcontroller.

Operators

Pin1 and Pin2 are Port.Pin constants that specify the output pins on the PICmicro™.

Note is a variable or constant specifying frequency in Hertz (Hz, 0 to 16000) of the tones.

Duration can be a variable or constant that determines how long the Notes are played. In approx 1ms increments (0 to 65535).

Example 1

```
' Generate a 2500Hz tone and a 3500Hz tone for 1 second.  
' The 2500Hz note is played from the first pin specified (PORTB.0),  
' and the 3500Hz note is played from the second pin specified (PORTB.1).  
Device = 16F877  
Declare Xtal = 20  
Symbol Pin1 = PORTB.0  
Symbol Pin2 = PORTB.1  
Sound2 Pin1, Pin2, [2500\3500\1000]  
Stop
```

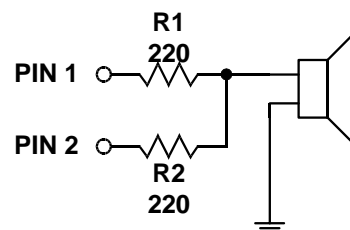
Example 2

```
' Play two sets of notes 2500Hz and 3500Hz for 1 second  
' and the second two notes, 2500Hz and 3500Hz for 2 seconds.  
Device = 16F877  
Declare Xtal = 20  
Symbol Pin1 = PORTB.0  
Symbol Pin2 = PORTB.1  
Sound2 Pin1, Pin2, [2500\3500 1000, 2500\3500\2000]  
Stop
```

Notes

Sound2 generates two pulses at the required frequency one on each pin specified. The **Sound2** command can be used to play tones through a speaker or audio amplifier. **Sound2** can also be used to play more complicated notes. By generating two frequencies on separate pins, a more defined sound can be produced. **Sound2** is somewhat dependent on the crystal frequency used for its note frequency, and duration.

Sound2 does not require any filtering on the output, and produces a cleaner note than **Freqout**. However, unlike **Freqout**, the note is not a SINE wave. See diagram: -



See also : **Freqout**, **DTMFout**, **Sound**.

Stop

Syntax Stop

Overview

Stop halts program execution by sending the microcontroller into an infinite loop.

Example

```
If A > 12 Then Stop  
{ code data }
```

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

Notes

Although **Stop** halts the microcontroller in its tracks it does not prevent any code listed in the BASIC source after it from being compiled.

See also : End, Sleep, Snooze.

Strn

Syntax

Strn *Byte Array* = *Item*

Overview

Load a **Byte Array** with null terminated data, which can be likened to creating a pseudo String variable.

Operators

Byte Array is the variable that will be loaded with values.

Item can be another **Strn** command, a **Str** command, **Str\$** command, or a quoted character string

Example

```
' Load the Byte Array String1 with null terminated characters

Include "Proton_4.Inc"      ' Demonstration based on the Proton dev board
Dim String1[21] as Byte    ' Create a Byte array with 21 elements

DelayMs 100                ' Wait for the LCD to stabilise
Cls                          ' Clear the LCD
Strn String1 = "HELLO WORLD"
' Load String1 with characters and null terminate it
Print Str String1         ' Display the string
Stop
```

See also: [Arrays as Strings](#), [Str\\$](#).

Str\$

Syntax

Str Byte Array = **Str\$** (Modifier Variable)

or

String = **Str\$** (Modifier Variable)

Overview

Convert a Decimal, Hex, Binary, or Floating Point value or variable into a null terminated string held in a **Byte array**, or a **String** variable. For use only with the **Str** and **Strn** commands, and real String variables.

Operators

Modifier is one of the standard modifiers used with **Print**, **Rsout**, **Hserout** etc. See list below. **Variable** is a variable that holds the value to convert. This may be a **Bit**, **Byte**, **Word**, **Dword**, or **Float**.

Byte Array must be of sufficient size to hold the resulting conversion and a terminating null character (0).

String must be of sufficient size to hold the resulting conversion.

Notice that there is no comma separating the Modifier from the Variable. This is because the compiler borrows the format and subroutines used in **Print**. Which is why the modifiers are the same: -

Bin{1..32}	Convert to binary digits
Dec{1..10}	Convert to decimal digits
Hex{1..8}	Convert to hexadecimal digits
Sbin{1..32}	Convert to signed binary digits
Sdec{1..10}	Convert to signed decimal digits
Shex{1..8}	Convert to signed hexadecimal digits
Ibin{1..32}	Convert to binary digits with a preceding '%' identifier
Idec{1..10}	Convert to decimal digits with a preceding '#' identifier
Ihex{1..8}	Convert to hexadecimal digits with a preceding '\$' identifier
ISbin{1..32}	Convert to signed binary digits with a preceding '%' identifier
ISdec{1..10}	Convert to signed decimal digits with a preceding '#' identifier
IShex{1..8}	Convert to signed hexadecimal digits with a preceding '\$' identifier

Example 1

```
' Convert a Word variable to a String of characters in a Byte array.
Include "Proton_4.Inc" ' Use the Proton board for the demo
' Create a byte array to hold converted value, and null terminator
Dim String1[11] as Byte
Dim Wrld as Word
DelayMs 100          ' Wait for the LCD to stabilise
Cls                  ' Clear the LCD
Wrld = 1234          ' Load the variable with a value
Strn String1 = Str$(Dec Wrld) ' Convert the Integer to a String
Print Str String1   ' Display the string
Stop
```

Example 2

```
' Convert a Dword variable to a String of characters in a Byte array.
Include "Proton_4.Inc"      ' Use the Proton board for the demo
' Create a byte array to hold converted value, and null terminator
Dim String1[11] as Byte
Dim Dwd1 as Dword
DelayMs 100                ' Wait for the LCD to stabilise
Cls                        ' Clear the LCD
Dwd1 = 1234                ' Load the variable with a value
Strn String1 = Str$(Dec Dwd1) ' Convert the Integer to a String
Print Str String1        ' Display the string
Stop
```

Example 3

```
' Convert a Float variable to a String of characters in a Byte array.
Include "Proton_4.Inc"      ' Use the Proton board for the demo
' Create a byte array to hold converted value, and null terminator
Dim String1[11] as Byte
Dim Flt1 as Float
DelayMs 100                ' Wait for the LCD to stabilise
Cls                        ' Clear the LCD
Flt1 = 3.14                ' Load the variable with a value
Strn String1 = Str$(Dec Flt1) ' Convert the Float to a String
Print Str String1        ' Display the string
Stop
```

Example 4

```
' Convert a Word variable to a Binary String of characters in an array.
Include "Proton_4.Inc"      ' Use the Proton board for the demo
' Create a byte array to hold converted value, and null terminator
Dim String1[34] as Byte
Dim Wrld1 as Word
DelayMs 100                ' Wait for the LCD to stabilise
Cls                        ' Clear the LCD
Wrld1 = 1234                ' Load the variable with a value
Strn String1 = Str$(Bin Wrld1) ' Convert the Integer to a String
Print Str String1        ' Display the string
Stop
```

If we examine the resulting string (Byte Array) converted with example 2, it will contain: -

character 1, character 2, character 3, character 4, 0

The zero is not character zero, but value zero. This is a null terminated string.

Notes

The **Byte Array** created to hold the resulting conversion, must be large enough to accommodate all the resulting digits, including a possible minus sign and preceding identifying character. %, \$, or # if the I version modifiers are used. The compiler will try and warn you if it thinks the array may not be large enough, but this is a rough guide, and you as the programmer must decide whether it is correct or not. If the size is not correct, any adjacent variables will be overwritten, with potentially catastrophic results.

See also : [Creating and using Strings, Strn, Arrays as Strings.](#)

Swap

Syntax

Swap *Variable, Variable*

Overview

Swap any two variable's values with each other.

Operators

Variable is the value to be swapped

Example

' If Dog = 2 and Cat = 10 then by using the swap command

' Dog will now equal 10 and Cat will equal 2.

Var1 = 10 *' Var1 equals 10*

Var2 = 20 *' Var2 equals 20*

Swap Var1, Var2 *' Var2 now equals 20 and Var1 now equals 10*

Symbol

Syntax

Symbol *Name* { = } *Value*

Overview

Assign an alias to a register, variable, or constant value

Operators

Name can be any valid identifier.

Value can be any previously declared variable, system register, or a Register.Bit combination. The equals '=' symbol is optional, and may be omitted if desired.

When creating a program it can be beneficial to use identifiers for certain values that don't change: -

```
Symbol Meter = 1
Symbol Centimetre = 100
Symbol Millimetre = 1000
```

This way you can keep your program very readable and if for some reason a constant changes later, you only have to make one change to the program to change all the values. Another good use of the constant is when you have values that are based on other values.

```
Symbol Meter = 1
Symbol Centimetre = Meter / 100
Symbol Millimetre = Centimetre / 10
```

In the example above you can see how the centimetre and millimetre were derived from the Meter.

Another use of the **Symbol** command is for assigning Port.Bit constants: -

```
Symbol LED = PORTA.0
High LED
```

In the above example, whenever the text LED is encountered, Bit-0 of PORTA is actually referenced.

Floating point constants may also be created using **Symbol** by simply adding a decimal point to a value.

```
Symbol PI = 3.14      ' Create a floating point constant named PI
Symbol FlNum = 5.0    ' Create a floating point constant with the value 5
```

Floating point constant can also be created using expressions.

```
' Create a floating point constant holding the result of the expression
Symbol Quanta = 5.0 / 1024
```

Notes

Symbol cannot create new variables, it simply aliases an identifier to a previously assigned variable, or assigns a constant to an identifier.

Toggle

Syntax

Toggle *Port.Bit*

Overview

Sets a pin to output mode and reverses the output state of the pin, changing 0 to 1 and 1 to 0.

Operators

Port.Bit can be any valid Port and Bit combination.

Example

```
High PORTB.0      ' Set bit 0 of PORTB high
Toggle PORTB.0    ' And now reverse the bit
Stop
```

See also : High, Low.

ToLower

Syntax

Destination String = **ToLower** (*Source String*)

Overview

Convert the characters from a source string to lower case.

Overview

Destination String can only be a **String** variable, and should be large enough to hold the correct amount of characters extracted from the *Source String*.

Source String can be a **String** variable, or a Quoted String of Characters. The *Source String* can also be a **Byte**, **Word**, **Dword**, **Float** or **Array**, variable, in which case the value contained within the variable is used as a pointer to the start of the Source String's address in RAM. A third possibility for *Source String* is a Label name, in which case a null terminated Quoted String of Characters is read from a **Cdata** table.

Example 1

' Convert the characters from SourceString to lowercase into DestString

```
Device = 18F452           ' A suitable device for Strings
Dim SourceString as String * 20 ' Create a String of 20 characters
Dim DestString as String * 20  ' Create another String

SourceString = "HELLO WORLD" ' Load the source string with characters
DestString = ToLower (SourceString) ' Convert to lowercase
Print DestString           ' Display the result, which will be "hello world"
Stop
```

Example 2

*' Convert the characters from a Quoted Character String to lowercase
' into DestString*

```
Device = 18F452           ' A suitable device for Strings
Dim DestString as String * 20 ' Create a String of 20 characters

DestString = ToLower ("HELLO WORLD") ' Convert to lowercase
Print DestString           ' Display the result, which will be "hello world"
Stop
```

Example 3

*' Convert to lowercase from SourceString into DestString using a pointer to
' SourceString*

```
Device = 18F452           ' A suitable device for Strings
Dim SourceString as String * 20 ' Create a String of 20 characters
Dim DestString as String * 20  ' Create another String
' Create a Word variable to hold the address of SourceString
Dim StringAddr as Word
```

```
SourceString = "HELLO WORLD"      ' Load the source string with characters
' Locate the start address of SourceString in RAM
StringAddr = AddressOf(SourceString)
DestString = ToLower(StringAddr) ' Convert to lowercase
Print DestString      ' Display the result, which will be "hello world"
Stop
```

Example 4

' Convert chars from a Cdata table to lowercase and place into DestString

```
Device = 18F452      ' A suitable device for Strings
Dim DestString as String * 20 ' Create a String of 20 characters

DestString = ToLower (Source) ' Convert to lowercase
Print DestString      ' Display the result, which will be "hello world"
Stop
```

' Create a null terminated string of characters in code memory

Source:

```
Cdata "HELLO WORLD", 0
```

See also : [Creating and using Strings](#), [Creating and using Virtual Strings with Cdata](#), [Cdata](#), [Len](#), [Left\\$](#), [Mid\\$](#), [Right\\$](#), [Str\\$](#), [ToUpper](#), [AddressOf](#).

ToUpper

Syntax

Destination String = **ToUpper** (*Source String*)

Overview

Convert the characters from a source string to UPPER case.

Overview

Destination String can only be a **String** variable, and should be large enough to hold the correct amount of characters extracted from the *Source String*.

Source String can be a **String** variable, or a Quoted String of Characters . The *Source String* can also be a **Byte**, **Word**, **Dword**, **Float** or **Array**, variable, in which case the value contained within the variable is used as a pointer to the start of the Source String's address in RAM. A third possibility for *Source String* is a Label name, in which case a null terminated Quoted String of Characters is read from a **Cdata** table.

Example 1

```
' Convert the characters from SourceString to UpperCase and place into
' DestString

Device = 18F452           ' A suitable device for Strings
Dim SourceString as String * 20 ' Create a String of 20 characters
Dim DestString as String * 20  ' Create another String

SourceString = "hello world" ' Load the source string with characters
DestString = ToUpper(SourceString) ' Convert to uppercase
Print DestString ' Display the result, which will be "HELLO WORLD"
Stop
```

Example 2

```
' Convert the chars from a Quoted Character String to UpperCase
' and place into DestString

Device = 18F452           ' A suitable device for Strings
Dim DestString as String * 20 ' Create a String of 20 characters

DestString = ToUpper("hello world") ' Convert to uppercase
Print DestString ' Display the result, which will be "HELLO WORLD"
Stop
```

Example 3

```
' Convert to UpperCase from SourceString into DestString using a pointer to
' SourceString

Device = 18F452           ' A suitable device for Strings
Dim SourceString as String * 20 ' Create a String of 20 characters
Dim DestString as String * 20  ' Create another String
' Create a Word variable to hold the address of SourceString
Dim StringAddr as Word
```

```
' Load the source string with characters
SourceString = "hello world"
' Locate the start address of SourceString in RAM
StringAddr = AddressOf(SourceString)
DestString = ToUpper(StringAddr) ' Convert to uppercase
Print DestString ' Display the result, which will be "HELLO WORLD"
Stop
```

Example 4

```
' Convert chars from Cdata table to uppercase and place into DestString
```

```
Device = 18F452 ' A suitable device for Strings
Dim DestString as String * 20 ' Create a String of 20 characters
```

```
DestString = ToUpper(Source) ' Convert to uppercase
Print DestString ' Display the result, which will be "HELLO WORLD"
Stop
```

```
' Create a null terminated string of characters in code memory
```

```
Source:
```

```
Cdata "hello world", 0
```

See also : [Creating and using Strings](#), [Creating and using Virtual Strings with Cdata](#), [Cdata](#), [Len](#), [Left\\$](#), [Mid\\$](#), [Right\\$](#), [Str\\$](#), [ToLower](#), [AddressOf](#) .

Toshiba_Command

Syntax

Toshiba_Command *Command, Value*

Overview

Send a command with or without parameters to a Toshiba T6963 graphic LCD.

Operators

Command can be a constant, variable, or expression, that contains the command to send to the LCD. This will always be an 8-bit value.

Value can be a constant, variable, or expression, that contains an 8-bit or 16-bit parameter associated with the command. An 8-bit value will be sent as a single parameter, while a 16-bit value will be sent as two parameters. Parameters are optional as some commands do not require any. Therefore if no parameters are included, only a command is sent to the LCD.

Because the size of the parameter is vital to the correct operation of specific commands, you can force the size of the parameter sent by issuing either the text “**Byte**” or “**Word**” prior to the parameter’s value.

```
Toshiba_Command $C0, Byte $FF01 ' Send the low byte of the 16-bit value.  
Toshiba_Command $C0, Word $01   ' Send a 16-bit value regardless.
```

The explanation of each command is too lengthy for this document, however they can be found in the Toshiba T6963 datasheet. The example program shown below contains a condensed list of commands.

Example

```
' Pan two pages of text left and right on a 128x64 Toshiba T6963 graphic LCD  
Device = 18F452  
Declare LCD_Type = Toshiba           ' Use a Toshiba T6963 graphic LCD  
,  
' LCD interface pin assignments  
,  
Declare LCD_DTPort = PORTD           ' LCD's Data port  
Declare LCD_WRPin = PORTE.2         ' LCD's WR line  
Declare LCD_RDPin = PORTE.1         ' LCD's RD line  
Declare LCD_CEPin = PORTE.0         ' LCD's CE line  
Declare LCD_CDPin = PORTA.1         ' LCD's CD line  
Declare LCD_RSTPin = PORTA.0        ' LCD's RESet line (Optional)  
,  
' LCD characteristics  
,  
Declare LCD_Text_Pages = 2           ' Choose two text pages  
Declare LCD_RAM_Size = 8192         ' Amount of RAM the LCD contains  
Declare LCD_X_Res = 128             ' LCD's X Resolution  
Declare LCD_Y_Res = 64              ' LCD's Y Resolution  
Declare LCD_Font_Width = 6          ' The width of the LCD's font  
Declare LCD_Text_Home_Address = 0   ' Text RAM starts at address 0
```

```
' LCD Display Constants: -
' Register set commands:
  Symbol T_Cursor_POINTER_Set = $21 ' Cursor Pointer Set
' Offset Register Set (CGRAM start address offset)
  Symbol T_OFFset_REG_Set = $22
  Symbol T_Adr_POINTER_Set = $24 ' Address Pointer Set
' Control Word Set commands:
  Symbol T_Text_Home_Set = $40 ' Text Home Address Set
  Symbol T_Text_AREA_Set = $41 ' Text Area Set
  Symbol T_GRAPH_Home_Set = $42 ' Graphics Home address Set
  Symbol T_GRAPH_AREA_Set = $43 ' Graphics Area Set
' Mode Set commands:
  Symbol T_or_MODE = $80 ' or mode
  Symbol T_xor_MODE = $81 ' xor mode
  Symbol T_and_MODE = $83 ' and mode
  Symbol T_Text_ATTR_MODE = $84 ' Text Attribute mode
  Symbol T_INT_CG_MODE = $80 ' Internal CG ROM mode
  Symbol T_EXT_CG_MODE = $88 ' External CG RAM mode
' Display Mode commands (or together required bits):
  Symbol T_DISPLAY_OFF = $90 ' Display off
  Symbol T_BLINK_ON = $91 ' Cursor Blink on
  Symbol T_Cursor_ON = $92 ' Cursor on
  Symbol T_Text_ON = $94 ' Text mode on
  Symbol T_Graphic_ON = $98 ' Graphic mode on
  Symbol T_Text_and_GRAPH_ON = $9C ' Text and graphic mode on
' Cursor Pattern Select:
  Symbol T_Cursor_1LINE = $A0 ' 1 line cursor
  Symbol T_Cursor_2LINE = $A1 ' 2 line cursor
  Symbol T_Cursor_3LINE = $A2 ' 3 line cursor
  Symbol T_Cursor_4LINE = $A3 ' 4 line cursor
  Symbol T_Cursor_5LINE = $A4 ' 5 line cursor
  Symbol T_Cursor_6LINE = $A5 ' 6 line cursor
  Symbol T_Cursor_7LINE = $A6 ' 7 line cursor
  Symbol T_Cursor_8LINE = $A7 ' 8 line cursor
' Data Auto Read/Write:
  Symbol T_Data_AUTO_WR = $B0 ' Data write with auto increment of address
  Symbol T_Data_AUTO_RD = $B1 ' Data read with auto increment of address
  Symbol T_AUTO_Data_Reset = $B2 ' Disable auto read/write
' Data Read/Write:
  Symbol T_Data_WR_Inc = $C0 ' Data write and increment address
  Symbol T_Data_RD_Inc = $C1 ' Data read and increment address
  Symbol T_Data_WR_Dec = $C2 ' Data write and decrement address
  Symbol T_Data_RD_Dec = $C3 ' Data read and decrement address
  Symbol T_Data_WR = $C4 ' Data write with no address change
  Symbol T_Data_RD = $C5 ' Data read with no address change
' Screen Peek:
  Symbol T_SCREEN_Peek = $E0 ' Read the display
' Screen Copy:
  Symbol T_SCREEN_COPY = $E8 ' Copy a line of the display
' Bit Set/Reset (or with bit number 0-7):
  Symbol T_Bit_Reset = $F0 ' Pixel clear
  Symbol T_Bit_Set = $F8 ' Pixel set
```

```
' Create two variables for the demonstration
Dim Pan_Loop as Byte      ' Holds the amount of pans to perform
Dim Ypos as Byte         ' Holds the Y position of the displayed text
,
' The Main program loop starts here
,
DelayMs 100              ' Wait for the LCD to stabilise
ADCON1 = 7              ' PORTA and PORTE to all digital mode
Cls                      ' Clear and initialise the LCD
' Place text on two screen pages
For Ypos = 1 to 6
    Print At Ypos, 0, "  THIS IS PAGE ONE          THIS IS PAGE TWO"
Next
' Draw a box around the display
Line 1, 0, 0, 127, 0    ' Top line
LineTo 1, 127, 63      ' Right line
LineTo 1, 0, 63        ' Bottom line
LineTo 1, 0, 0         ' Left line
' Pan from one screen to the next then back
While                   ' Create an infinite loop
    For Pan_Loop = 0 to 22
        ' Increment the Text home address
        Toshiba_Command T_Text_Home_Set, Word Pan_Loop
        DelayMs 200
    Next
    DelayMs 200
    For PAN_Loop = 22 to 0 Step -1
        ' Decrement the Text home address
        Toshiba_Command T_Text_Home_Set, Word Pan_Loop
        DelayMs 200
    Next
    DelayMs 200
Wend                    ' Do it forever
```

Notes

When the Toshiba LCD's **Declares** are issued within the BASIC program, several internal variables and constants are automatically created that contain the Port and Bits used by the actual interface and also some constant values holding valuable information concerning the LCD's RAM boundaries and setup. These variables and constants can be used within the BASIC or Assembler environment. The internal variables and constants are: -

Variables.

__LCD_DTPort	The Port where the LCD's data lines are attached.
__LCD_WRPot	The Port where the LCD's WR pin is attached.
__LCD_RDPot	The Port where the LCD's RD pin is attached.
__LCD_CEPot	The Port where the LCD's CE pin is attached.
__LCD_CDPot	The Port where the LCD's CD pin is attached.
__LCD_RSTPot	The Port where the LCD's RST pin is attached.

Constants.

<code>__LCD_Type</code>	The type of LCD targeted. 0 = Alphanumeric, 1 = Samsung, 2 = Toshiba.
<code>__LCD_WRPin</code>	The Pin where the LCD's WR line is attached.
<code>__LCD_RDPin</code>	The Pin where the LCD's RD line is attached.
<code>__LCD_CEPin</code>	The Pin where the LCD's CE line is attached.
<code>__LCD_CDPin</code>	The Pin where the LCD's CD line is attached.
<code>__LCD_RSTPin</code>	The Pin where the LCD's RST line is attached.
<code>__LCD_Text_Pages</code>	The amount of TEXT pages chosen.
<code>__LCD_Graphic_Pages</code>	The amount of Graphic pages chosen.
<code>__LCD_RAM_Size</code>	The amount of RAM that the LCD contains.
<code>__LCD_X_Res</code>	The X resolution of the LCD. i.e. Horizontal pixels.
<code>__LCD_Y_Res</code>	The Y resolution of the LCD. i.e. Vertical pixels.
<code>__LCD_Font_Width</code>	The width of the font. i.e. 6 or 8.
<code>__LCD_Text_AREA</code>	The amount of characters on a single line of TEXT RAM.
<code>__LCD_Graphic_AREA</code>	The amount of characters on a single line of Graphic RAM.
<code>__LCD_Text_Home_Address</code>	The Starting address of the TEXT RAM.
<code>__LCD_Graphic_Home_Address</code>	The Starting address of the Graphic RAM.
<code>__LCD_CGRAM_Home_Address</code>	The Starting address of the CG RAM.
<code>__LCD_End_OF_Graphic_RAM</code>	The Ending address of Graphic RAM.
<code>__LCD_CGRAM_OFFSET</code>	The Offset value for use with CG RAM.

Notice that each name has TWO underscores preceding it. This should ensure that duplicate names are not defined within the BASIC environment.

It may not be apparent straight away why the variables and constants are required, however, the Toshiba LCDs are capable of many tricks such as panning, page flipping, text manipulation etc, and all these require some knowledge of RAM boundaries and specific values relating to the resolution of the LCD used.

See also : `LCDRead`, `LCDWrite`, `Pixel`, `Plot`, `Toshiba_UDG`, `UnPlot`.
See Print for circuit.

Toshiba_UDG

Syntax

Toshiba_UDG *Character*, [*Value* {, *Values* }]

Overview

Create **User Defined Graphics** for a Toshiba T6963 graphic LCD.

Operators

Character can be a constant, variable, or expression, that contains the character to define. User defined characters start from 160 to 255.

Values is a list of constants, variables, or expressions, that contain the information to build the User Defined character. There are also some modifiers that can be used in order to access UDG data from various tables.

Example

```
' Create four User Defined Characters using four different methods
Device = 18F452
Declare Xtal = 20

Declare LCD_Type = Toshiba      ' Use a Toshiba T6963 graphic LCD
,
' LCD interface pin assignments
,
Declare LCD_DTPort = PORTD      ' LCD's Data port
Declare LCD_WRPin = PORTE.2    ' LCD's WR line
Declare LCD_RDPin = PORTE.1    ' LCD's RD line
Declare LCD_CEPin = PORTE.0    ' LCD's CE line
Declare LCD_CDPin = PORTA.1    ' LCD's CD line
Declare LCD_RSTPin = PORTA.0   ' LCD's RESet line (Optional)
,
' LCD characteristics
,
Declare LCD_X_Res = 128        ' LCD's X Resolution
Declare LCD_Y_Res = 64        ' LCD's Y Resolution
Declare LCD_Font_Width = 8    ' The width of the LCD's font
Dim UDG_3[8] as Byte          ' Create a byte array to hold UDG data
Dim DemoChar as Byte         ' Create a variable for the demo loop
' Create some User Defined Graphic data in eeprom memory
UDG_1 Edata $18, $18, $3C, $7E, $DB, $99, $18, $18
,
' The main demo loop starts here
DelayMs 100                   ' Wait for the LCD to stabilise
ADCON1 = 7                    ' PORTA and PORTE to all digital mode
Cls                            ' Clear both text and graphics of the LCD
' Load the array with UDG data
Str UDG_3 = $18, $18, $99, $DB, $7E, $3C, $18, $18
,
' Print user defined graphic chars 160, 161, 162, and 162 on the LCD
,
Print At 1, 0, "Char 160 = ", 160
Print At 2, 0, "Char 161 = ", 161
Print At 3, 0, "Char 162 = ", 162
Print At 4, 0, "Char 163 = ", 163
```

```

Toshiba_UDG 160, [Estr UDG_1]      ' Place UDG edata into character 160
Toshiba_UDG 161, [UDG_2]          ' Place UDG cdata into character 161
Toshiba_UDG 162, [Str UDG_3\8]    ' Place UDG array into character 162
' Place values into character 163
Toshiba_UDG 163, $0C, $18, $30, $FF, $FF, $30, $18, $0C]
While                               ' Create an infinite loop
  For DemoChar = 160 to 163         ' Cycle through characters 160 to 163
    Print At 0, 0, DemoChar        ' Display the character
    DelayMs 200                    ' A small delay
  Next                               ' Close the loop
Wend                                 ' Do it forever
'
' Create some User Defined Graphic data in code memory
UDG_2: Cdata $30, $18, $0C, $FF, $FF, $0C, $18, $30

```

Notes

User Defined Graphic values can be stored in on-board eeprom memory by the use of **Edata** tables, and retrieved by the use of the **Estr** modifier. Eight, and only Eight, values will be read with a single **Estr**:

```

UDG_1 Edata $18, $18, $3C, $7E, $DB, $99, $18, $18
  Toshiba_UDG 160, [Estr UDG_1]

```

User Defined Graphic values can also be stored in code memory, on devices that can access their own code memory, and retrieved by the use of a label name associated with a **Cdata** table. Eight, and only Eight, values will be read with a single label name:

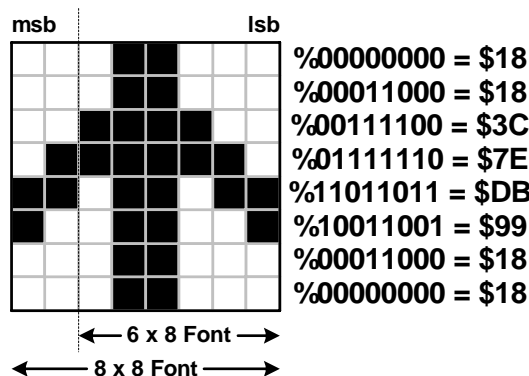
```

  Toshiba_UDG 161, [UDG_2]
UDG_2:
  Cdata $30, $18, $0C, $FF, $FF, $0C, $18, $30

```

The use of the **Str** modifier will retrieve values stored in an array, however, this is not recommended as it will waste precious RAM.

The Toshiba LCD's font is designed in an 8x8 grid or a 6x8 grid depending on the font size chosen. The diagram below shows a designed character and its associated values.



See also : LCDRead, LCDWrite, Pixel, Plot, Toshiba_Command, UnPlot.
See Print for circuit.

UnPlot

Syntax

UnPlot *Ypos, Xpos*

Overview

Clear an individual pixel on a graphic LCD.

Operators

Xpos can be a constant, variable, or expression, pointing to the X-axis location of the pixel to clear. This must be a value of 0 to the X resolution of the LCD. Where 0 is the far left row of pixels.

Ypos can be a constant, variable, or expression, pointing to the Y-axis location of the pixel to clear. This must be a value of 0 to the Y resolution of the LCD. Where 0 is the top column of pixels.

Example

```
Device = 16F877
Declare Xtal = 4

Declare LCD_Type = Graphic      ' Use a Samsung KS0108 Graphic LCD
,
' Graphic LCD Pin Assignments
,
Declare LCD_DTPort = PORTD
Declare LCD_RSPin = PORTC.2
Declare LCD_RWPin = PORTE.0
Declare LCD_ENPin = PORTC.5
Declare LCD_CS1Pin = PORTE.1
Declare LCD_CS2Pin = PORTE.2

Dim Xpos as Byte
ADCON1 = 7                      ' PORTA and PORTE to all digital mode
Cls                              ' Clear the LCD
,
' Draw a line across the LCD
,
While                            ' Create an infinite loop
  For Xpos = 0 to 127
    Plot 20, Xpos
    DelayMs 10
  Next
,
' Now erase the line
,
  For Xpos = 0 to 127
    UnPlot 20, Xpos
    DelayMs 10
  Next
Wend
```

See also : **LCDRead, LCDWrite, Pixel, Plot. See Print for circuit.**

USBinit

Syntax USBinit

Overview

Initialise the USB peripheral and wait until the USB bus is configured and enabled.

Notes.

USBinit is optional within the BASIC program itself. If it is not used in the program, the compiler will initialise the USB peripheral itself before the program starts.

If the device contains the OSCTUNE register, it will set bit PLEN and enable the x4 PLL.

The benefit of adding **USBinit** within the BASIC program is that you will have the opportunity to set or clear this bit as required.

See also : **USBout, USBin, USBpoll, Config_Start...Config_End.**

USBIn

Syntax

USBIn *Endpoint, Buffer, Countvar, Label*

Overview

Receive USB data from the host computer and place it into *Buffer*.

Operators

Endpoint is a constant value (0 - 15) that indicates which EndPoint to receive data from.

Buffer is a Byte array or String that will contain the bytes received. This may be up to 128 bytes in length if using CDC and 64 bytes for HID.

Countvar is a constant, variable or expression that indicates how many bytes are transferred from the *Buffer*. The text **Auto** may be placed instead of the *Countvar* parameter. This will configure the receiving bus to it's maximum, which is 128 bytes for CDC and 64 bytes for HID.

Label is an optional valid BASIC label, that **USBIn** will jump to in the event that no data is available.

Example 1

```
' USB interface
  Dim Buffer[8] as Byte
Try_Again:
  USBIn 1, Buffer, 4, Try_Again
```

Example 2

```
' USB demo program for CDC virtual serial emulation
' Wait for a byte from USB and transmit several characters
,
  Declare Reminders = Off
  Device = 18F26J50
  Declare Xtal = 48
  Declare Optimiser_Level = 1

  Include "CDC_Descriptor.inc"           ' Include the CDC descriptors

  Dim Byteout As Byte = $41
  Dim Wordout As Word = $4142
  Dim DWordout As Dword = $41424344
  Dim RxBuffer[16] As Byte
  Dim TxBuffer As String * 16 = "Hello World\r"
  Dim CodeText As Code = "Hello World\r"

  OSCTUNE.6 = 1                         ' Enable PLL for 18F87J50 family
  DelayMS 10
  USBInit                                ' Initialise USB

  While
IdleLoop:
  USBIn 3, RxBuffer, 16, IdleLoop        ' Wait for USB input
' Transmit to a serial terminal
OutLoop1:
  USBOut 3, CodeText, Auto, OutLoop1
OutLoop2:
  USBOut 3, TxBuffer, Auto, OutLoop2
OutLoop3:
  USBOut 3, Byteout, 1, OutLoop3
```

```
OutLoop4:
    USBOut 3, Wordout, 2, OutLoop4
OutLoop5:
    USBOut 3, DWordout, 4, OutLoop5
Wend
' Wait for next buffer
' Configure the 18F26J50 for 48MHz operation using a 12MHz crystal
Config_Start
CPUDIV = OSC1      ' No CPU System clock divide
PLLDIV = 3         ' Divide by 3 (12 MHz oscillator input)
OSC = HSPLL        ' HS PLL oscillator x 4
CP0 = OFF          ' Program memory is not code-protected
WDTEN = OFF        ' Watchdog disabled
DEBUG = OFF        ' Hardware Debug disabled
XINST = OFF        ' Extended Instruction Set: Disabled
T1DIG = OFF        ' Secondary Oscillator clock source may not be selected
LPT1OSC = ON       ' Timer1 Oscillator: Low-power operation
FCMEN = OFF        ' Fail-Safe Clock Monitor: Disabled
IESO = OFF         ' Internal External Oscillator Switch Over Mode: Disable
WDTPS = 128        ' Watchdog Postscaler: 1:128
DSWDTOSC = INTOSCREF ' DSWDT uses INTRC
RTCOSC = INTOSCREF ' RTCC Clock Select: RTCC uses INTRC
DSBOREN = OFF      ' Deep Sleep BOR Disabled
DSWDTEN = OFF      ' Deep Sleep Watchdog Timer: Disabled
DSWDTPS = 128      ' Deep Sleep Watchdog Postscaler 1:128 (132 ms)
IOL1WAY = OFF      ' The IOLOCK bit can be set and cleared as needed
MSSP7B_EN = MSK5   ' 5 Bit address masking mode
WPCFG = OFF        ' Configuration Words page not erase/write-protected
WPDIS = OFF        ' WFPF<5:0>/WPEND region ignored
Config_End
```

Two USB interface types have been implemented within the compiler; HID (Human Interface Device) and CDC (Communication Device Class). These are chosen by the type of descriptor used. For example, the **CDC_Descriptor.Inc** descriptor file will use the CDC interface, while **HID_Descriptor.Inc** will use the HID interface. Both these files can be found within the compiler's Includes\Sources folder.

The **USBin** command polls the USB interface before continuing, therefore there is not always a need to use the **USBpoll** command.

The *Label* part of the **USBin** command is optional and can be omitted if required. Instead, the Carry flag (STATUS.0) can be checked to see if the microcontroller or USB transceiver has control of the Dual Port RAM buffer. The Carry will return clear if the microcontroller has control of the buffer and is able to receive some data.

Upon exiting the **USBin** command, register PRODL will contain the amount of bytes received.

Notes.

The method used for USB is polled, meaning that no interrupt is working in the background. However, this does mean that either a **USBpoll**, **USBin**, or **USBout** command needs to be executed approximately every 10ms or the USB interface connection will be lost.

USB must work at an oscillator speed of 48MHz. Achieving this frequency is accomplished by the use of the device's divide and multiply configuration fuse settings. See the relevant data-sheet for more information concerning these.

See also : **USBinit, USBout, USBpoll, Config_Start...Config_End.**

USBout

Syntax

USBout *Endpoint, Buffer, Countvar, Label*

Overview

Take **Countvar** number of bytes from **Buffer** and send them to the USB **Endpoint**.

Operators

Endpoint is a constant value (0 - 15) that indicates which EndPoint to transmit data from.

Buffer can be any of the compiler's variable or constant types, and contains the bytes to transmit. This may be up to 128 bytes in length if using CDC and 64 bytes for HID.

Countvar is a constant, variable or expression that indicates how many bytes are transferred from the **Buffer**. The text **Auto** may be placed instead of the **Countvar** value, which will transmit data until a null is found, or until the correct amount of bytes are transmitted for the variable size.

Label is an optional valid BASIC label, that **USBout** will jump to in the event that no data is available.

Example

```
Dim Buffer[8] as Byte
Try_Again:
USBout 1, Buffer, 4, Try_Again
```

Notes.

The **Label** used for buffer control may be omitted and the microcontroller's Carry flag (STATUS.0) monitored instead:-

```
Repeat
  USBout 3, USB_BUFFER, 4 ' Transmit 4 bytes from USB_BUFFER
Until STATUS.0 = 0       ' Wait for control over the buffer RAM
```

The **CountVar** parameter can also be replaced with the text **Auto**, in which case a string of characters terminated by a null (0) will be transmitted, or the amount of bytes that a particular variable type used will be transmitted. The **Countvar** parameter can be omitted, in which case **Auto** is implied: -

```
USBout 3, "Hello Wordl\n\r"
```

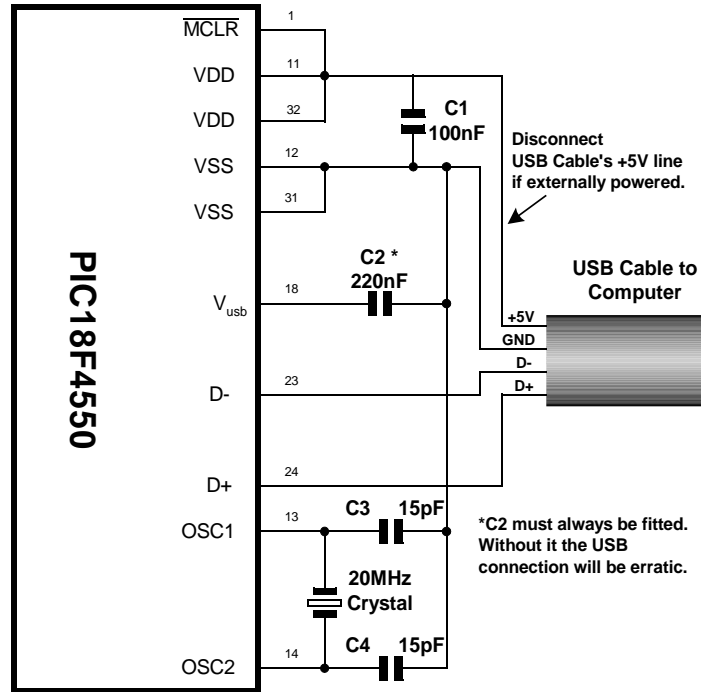
The buffer itself can take the form of any variable type of the compiler, and even the internal USB buffer itself. The internal USB buffer is brought into the BASIC code named **__USBout_Buffer** (note the two preceding underscores).

In the event that **Auto** has been used for the **Countvar** parameter, a **Bit** or **Byte** variable will transmit 1 byte of data, a **Word** will transmit 2 bytes (lowest byte first), a **Dword** and **Float** will transmit 4 bytes of data (lowest byte first).

The method used for USB is polled, meaning that no interrupt is working in the background. However, this does mean that either a **USBpoll**, **USBin**, or **USBout** command needs to be executed approximately every 10ms for HID and 5ms for CDC or the USB interface connection will be lost.

The USB library subroutines require the use of the DP (Dual Port) RAM starting at address \$0200 or \$0400 depending on the device used. This leaves the RAM underneath DP RAM available for the BASIC program. However, the DP USB buffers can also be accessed directly from BASIC. `__USBOUT_BUFFER`, and `__USBIN_BUFFER` are declared automatically as **String** type variables within the `USB_Mem.inc` file, located within the compiler `Includes\Sources` folder.

USB must work at an oscillator speed of 48MHz. Achieving this frequency is accomplished by the use of the device's divide and multiply configuration fuse settings. See the relevant data-sheet for more information concerning these.



Typical circuit for self powered USB interface.

The **USBout** command polls the USB interface before transferring its data to the bus, and returns with the Carry flag (`STATUS.0`) clear if it has control over the Dual Port buffer.

```
Repeat
  USBout 3, __USBout_BUFFER, Auto
Until STATUS.0 = 0
```

Two USB interface types have been implemented within the compiler; HID (Human Interface Device) and CDC (Communication Device Class). These are chosen by the type of descriptor used. For example, the **CDC_Descriptor.Inc** descriptor file will use the CDC interface, while **HID_Descriptor.Inc** will use the HID interface. Both these files can be found within the compiler's `Includes\Sources` folder.

Example.

```
' Demonstrate a HID (Human Interface Device) interface
' When connected to the PC, the mouse pointer will rotate in a small square

Device = 18F4550           ' Choose a device with on-board full speed USB
Declare Xtal = 48          ' Inform the compiler we're operating at 48MHz
Declare Optimiser_Level = 1
Declare Dead_Code_Remove = On

Include "HID_Descriptor.inc" ' Include the HID descriptors

Dim Buffer[4] As Byte
Dim Loop_Count As Byte
Dim Position As Byte
' High if microcontroller does not have control over the DP Buffer
Symbol Carry_Flag = STATUS.0

-----
' The main program loop starts here
DelayMS 10                 ' Wait for things to stabilise
Clear Buffer                ' Clear the array before we start

Repeat
  USBPoll                  ' Wait for USB to become attached
Until USB_tConnected = 1 Or USB_tConfigured = 1
  While
    For Position = 0 To 3   ' Move through each position
      For Loop_Count = 0 To 31 ' 32 steps in each direction
        Select Position
          Case 0             ' Move Up?
            Buffer#1 = 0
            Buffer#2 = -2
          Case 1             ' Move Right?
            Buffer#1 = 2
            Buffer#2 = 0
          Case 2             ' Move Down?
            Buffer#1 = 0
            Buffer#2 = 2
          Case 3             ' Move Left?
            Buffer#1 = -2
            Buffer#2 = 0
        EndSelect
        Repeat
          USBOut 1, Buffer, 4 ' Send the Buffer to endpoint 1
        Until Carry_Flag = 0 ' Keep trying if we don't have control
      Next
    Next
  Next
Wend

-----
' Configure the 18F4550 for 48MHz operation using a 12MHz crystal
Config_Start
PLLDIV = 3                 ' Divide by 3 (12 MHz oscillator input)
CPUDIV = OSC1_PLL2        ' [OSC1/OSC2 Src: /1][96 MHz PLL Src: /2]
USBDIV = 1                 ' USB clock source comes directly from the primary osc
FOSC = HSPLL_HS           ' HS oscillator, PLL enabled
FCMEN = OFF               ' Fail-Safe Clock Monitor disabled
IESO = OFF                 ' Oscillator Switchover mode disabled
```

```
PWRT = ON      ' PWRT enabled
BOR = ON      ' Brown-out Reset enabled in hardware only
BORV = 3      ' Brown-out Voltage bits: Minimum setting
VREGEN = ON   ' USB voltage regulator enabled
WDT = OFF     ' Watchdog Timer Disabled - SW Controlled
WDTPS = 128   ' Watchdog Timer Postscale Select bits: 1:128
MCLRE = ON    ' MCLR pin enabled, RE3 input pin disabled
LPT1OSC = ON  ' Timer1 configured for low-power operation
PBADEN = OFF  ' PORTB<4:0> pins are configured as digital I/O on Reset
CCP2MX = ON   ' CCP2 input/output is multiplexed with RC1
STVREN = OFF  ' Stack full/underflow will not cause Reset
LVP = OFF     ' Single-Supply ICSP disabled
XINST = OFF   ' Instruction set extension disabled
DEBUG = OFF   ' Background debugger disabled
CP0 = OFF     ' Block 0 (000800-001FFFh) not code-protected
CP1 = OFF     ' Block 1 (002000-003FFFh) not code-protected
CP2 = OFF     ' Block 2 (004000-005FFFh) not code-protected
CP3 = OFF     ' Block 3 (006000-007FFFh) not code-protected
CPB = OFF     ' Boot block (000000-0007FFFh) not code-protected
CPD = OFF     ' Data EEPROM not code-protected
WRT0 = OFF    ' Block 0 (000800-001FFFh) not write-protected
WRT1 = OFF    ' Block 1 (002000-003FFFh) not write-protected
WRT2 = OFF    ' Block 2 (004000-005FFFh) not write-protected
WRT3 = OFF    ' Block 3 (006000-007FFFh) not write-protected
WRTB = OFF    ' Boot block (000000-0007FFFh) not write-protected
WRTC = OFF    ' Config registers (300000-3000FFFh) not write-protected
WRD = OFF     ' Data eeprom not write-protected
EBTR0 = OFF   ' Block 0 not protected from table reads
EBTR1 = OFF   ' Block 1 not protected
EBTR2 = OFF   ' Block 2 not protected
EBTR3 = OFF   ' Block 3 not protected
EBTRB = OFF   ' Boot block not protected
```

Config_End

See also : USBinit, USBin, USBpoll, Config_Start...Config_End.

USBpoll

Syntax USBpoll

Overview

Poll the USB interface in order to keep it attached to the bus.

Notes

If the commands **USBin** or **USBout** are not used within a program loop, the interface will drop off the bus, therefore issue the **USBpoll** command to stop this happening. This command should be issued at least every 10ms if using a HID interface and at least once every 5ms for a CDC interface.

Upon exiting the **USBpoll** command, the state of the bus can be checked via the USB variable `__USB_bDeviceState`. This variable resides in a higher RAM bank of the PICmicro™ (as do all of the USB variables), which means that bank switching will take place whenever it is accessed. For this reason, the **USBpoll** subroutine loads this variable into a variable within Access RAM. The variable it uses is named `USB_bStatus`.

Several states are declared within the `USB_Dev.Inc` file located within the compiler's Includes folder. These are: -

<code>DETACHED_STATE</code>	<code>0</code>
<code>ATTACHED_STATE</code>	<code>1</code>
<code>POWERED_STATE</code>	<code>2</code>
<code>DEFAULT_STATE</code>	<code>4</code>
<code>ADDRESS_PENDING_STATE</code>	<code>8</code>
<code>ADDRESS_STATE</code>	<code>16</code>
<code>CONFIGURED_STATE</code>	<code>32</code>

Within the `USB_Mem.inc` file, there are several bits pre-declared for each of the above states. The relevant ones are:

```
USB_tConnected      ' Set if the USB is connected
USB_tConfigured     ' Set if the USB is Configured
```

Example

```
' Wait the for USB interface to be recognised and attached
Repeat
    USBpoll
Until USB_tConnected = 1 Or USB_tConfigured = 1
```

With newer devices, testing the `USB_tConnected` bit is all that is required, however, for older types such as the 18F4550, the `USB_tConfigured` bit has to be tested. For good measure, it may be prudent to test both of them.

See also : **USBinit, USBout, USBin, Config_Start...Config_End.**

Val

Syntax

Variable = **Val** (Array Variable, Modifier)

Overview

Convert a Byte Array or String containing Decimal, Hex, or Binary numeric text into it's integer equivalent.

Operators

Array Variable is a byte array or string containing the alphanumeric digits to convert and terminated by a null (i.e. value 0).

Modifier can be Hex, Dec, or Bin. To convert a Hex string, use the Hex modifier, for Binary, use the Bin modifier, for Decimal use the Dec modifier.

Variable is a variable that will contain the converted value. Floating point characters and variables cannot be converted, and will be rounded down to the nearest integer value.

Example 1

```
' Convert a string of hexadecimal characters to an integer
Include "Proton_4.Inc"      ' Use the Proton board for the demo
Dim String1[10] as Byte    ' Create a byte array as a String
Dim Wrd1 as Word           ' Create a variable to hold result
DelayMs 100                ' Wait for the LCD to stabilise
Cls                         ' Clear the LCD
Str String1 = "12AF",0     ' Load the String with Hex digits
Wrd1 = Val(String1,Hex)    ' Convert the String into an integer
Print Hex Wrd1             ' Display the integer as Hex
Stop
```

Example 2

```
' Convert a string of decimal characters to an integer
Include "Proton_4.Inc"      ' Use the Proton board for the demo
Dim String1[10] as Byte    ' Create a byte array as a String
Dim Wrd1 as Word           ' Create a variable to hold result
DelayMs 100                ' Wait for the LCD to stabilise
Cls                         ' Clear the LCD
Str String1 = "1234",0     ' Load the String with Decimal digits
Wrd1 = Val(String1,Dec)    ' Convert the String into an integer
Print Dec Wrd1             ' Display the integer as Decimal
Stop
```

Example 3

```
' Convert a string of binary characters to an integer
Include "Proton_4.Inc"      ' Use the Proton board for the demo
Dim String1[17] as Byte    ' Create a byte array as a String
Dim Wrd1 as Word           ' Create a variable to hold result
DelayMs 100                ' Wait for the LCD to stabilise
Cls                         ' Clear the LCD
Str String1 = "101010101000000",0 ' Load the String with Binary
Wrd1 = Val(String1,Bin)    ' Convert the String into an integer
Print Bin Wrd1             ' Display the integer as Binary
Stop
```

Notes

There are limitations with the **Val** command when used on a 14-bit core device, in that the array must fit into a single RAM bank. But this is not really a problem, just a little thought when placing the variables will suffice. The compiler will inform you if the array is not fully located inside a Bank, and therefore not suitable for use with the **Val** command.

This is not a problem with 18F devices, as they are able to access all their memory very easily.

The **Val** command is not recommended inside an expression, as the results are not predictable. However, the **Val** command can be used within an **If-Then**, **While-Wend**, or **Repeat-Until** construct, but the code produced is not as efficient as using it outside a construct, because the compiler must assume a worst case scenario, and use **Dword** comparisons.

```
Include "Proton_4.Inc"           ' Use the Proton board for the demo
Dim String1[10] as Byte         ' Create a byte array as a String
DelayMs 100                     ' Wait for the LCD to stabilise
Cls                             ' Clear the LCD
Str String1 = "123",0          ' Load the String with Dec digits
If Val(String1,Hex) = 123 Then  ' Compare the result
    Print At 1,1,Dec Val (String1,Hex)
Else
    Print At 1,1,"not Equal"
EndIf
Stop
```

See also: **Str, Strn, Str\$.**

AddressOf or VarPtr

Syntax

Assignment Variable = **VarPtr** (*Variable*)

or

Assignment Variable = **AddressOf** (*Variable*)

Overview

Returns the address of the variable in RAM, or a label in code memory. Commonly known as a pointer.

Operators

Assignment Variable can be any of the compiler's variable types, and will receive the pointer to the *variable's* address.

Variable can be any variable name used in the BASIC program.

Notes

Be careful if using **AddressOf** to locate the starting address of an array when using a standard 14-bit device, as arrays can cross bank boundaries, and the finishing address of the array may be in a different bank to its start address. The compiler can track bank changes internally when accessing arrays, but BASIC code generally cannot. For example, the most common use for **AddressOf** is when implementing indirect addressing using the microcontroller's FSR and INDF registers.

This is not the case with 18F devices, as the FSR0, 1, and 2 registers can access all memory areas linearly.

When using **AddressOf** with an enhanced 14-bit core device it will return the address of a variable or label plus the offsets required to make them linearly accessible. i.e. \$2000 for RAM and \$8000 for code memory.

While...Wend

Syntax

```
While Condition  
Instructions  
Instructions  
Wend
```

or

```
While Condition { Instructions : } Wend
```

Overview

Execute a block of instructions while a condition is true.

Example

```
Var1 = 1  
While Var1 <= 10  
    Print Dec Var1, " "  
    Var1 = Var1 + 1  
Wend
```

or

```
While PORTA.0 = 1: Wend ' Wait for a change on the Port
```

Notes

While-Wend, repeatedly executes *Instructions* **While** *Condition* is true. When the *Condition* is no longer true, execution continues at the statement following the **Wend**. *Condition* may be any comparison expression.

See also : **If-Then, Repeat-Until, For-Next.**

Xin

Syntax

Xin *DataPin*, *ZeroPin*, {*Timeout*, *Timeout Label*}, [*Variable*{,...}]

Overview

Receive X-10 data and store the House Code and Key Code in a variable.

Operators

DataPin is a constant (0 - 15), Port.Bit, or variable, that receives the data from an X-10 interface. This pin is automatically made an input to receive data, and should be pulled up to 5 Volts with a 4.7KΩ resistor.

ZeroPin is a constant (0 - 15), Port.Bit, or variable, that is used to synchronise to a zero-cross event. This pin is automatically made an input to received the zero crossing timing, and should also be pulled up to 5 Volts with a 4.7KΩ resistor.

Timeout is an optional value that allows program continuation if X-10 data is not received within a certain length of time. Timeout is specified in AC power line half-cycles (approximately 8.33 milliseconds).

Timeout Label is where the program will jump to upon a timeout.

Example

```
Dim HouseKey as Word
Cls
Loop:
' Receive X-10 data, go to NoData if none
Xin PORTA.2, PORTA.0, 10, NoData, [HouseKey]
' Display X-10 data on an LCD
Print At 1, 1, "House=", Dec HouseKey.Byte1, "Key=", Dec HouseKey.Byte0
Goto Loop          ' Do it forever
NoData:
Print "No Data"
Stop
```

Xout and Xin Declares

In order to make the **Xin** command's results more in keeping with the BASIC Stamp interpreter, two declares have been included for both **Xin** and Xout These are.

```
Declare Xout_Translate = On/Off, True/False or 1/0
```

and

```
Declare Xin_Translate = On/Off, True/False or 1/0
```

Notes

Xin processes data at each zero crossing of the AC power line as received on **ZeroPin**. If there are no transitions on this line, **Xin** will effectively wait forever.

Xin is used to receive information from X-10 devices that can transmit the appropriate data. X-10 modules are available from a wide variety of sources under several trade names. An interface is required to connect the microcontroller to the AC power line. The TW-523 for two-way X-10 communications is required by **Xin**. This device contains the power line interface and isolates the microcontroller from the AC line.

If **Variable** is a **Word** sized variable, then each House Code received will be stored in the upper 8-bits of the **Word** And each received Key Code will be stored in the lower 8-bits of the **Word** variable. If **Variable** is a **Byte** sized variable, then only the Key Code will be stored.

The House Code is a number between 0 and 15 that corresponds to the House Code set on the X-10 module A through P.

The Key Code can be either the number of a specific X-10 module or the function that is to be performed by a module. In normal operation, a command is first sent, specifying the X-10 module number, followed by a command specifying the desired function. Some functions operate on all modules at once so the module number is unnecessary. Key Code numbers 0-15 correspond to module numbers 1-16.

Warning. Under no circumstances should the microcontroller be connected directly to the AC power line. Voltage potentials carried by the power line will not only instantly destroy the microcontroller, but could also pose a serious health hazard.

See also : Xout.

Xout

Syntax

Xout *DataPin*, *ZeroPin*, [*HouseCode**KeyCode* {\Repeat} {, ...}]

Overview

Transmit a HouseCode followed by a KeyCode in X-10 format.

Operators

DataPin is a constant (0 - 15), Port.Bit, or variable, that transmits the data to an X-10 interface. This pin is automatically made an output.

ZeroPin is a constant (0 - 15), Port.Bit, or variable, that is used to synchronise to a zero-cross event. This pin is automatically made an input to received the zero crossing timing, and should also be pulled up to 5 Volts with a 4.7KΩ resistor.

HouseCode is a number between 0 and 15 that corresponds to the House Code set on the X-10 module A through P. The proper HouseCode must be sent as part of each command.

KeyCode can be either the number of a specific X-10 module, or the function that is to be performed by a module. In normal use, a command is first sent specifying the X-10 module number, followed by a command specifying the function required. Some functions operate on all modules at once so the module number is unnecessary. KeyCode numbers 0-15 correspond to module numbers 1-16.

Repeat is an optional operator, and if it is not included, then a repeat of 2 times (the minimum) is assumed. **Repeat** is normally reserved for use with the X-10 Bright and Dim commands.

Example

```
Dim House as Byte
Dim Unit as Byte
' Create some aliases of the keycodes
Symbol UnitOn = %10010      ' Turn module on
Symbol UnitOff = %11010    ' Turn module off
Symbol UnitsOff = %11100   ' Turn all modules off
Symbol LightsOn = %10100   ' Turn all light modules on
Symbol LightsOff = %10000  ' Turn all light modules off
Symbol Bright = %10110     ' Brighten light module
Symbol DimIt = %11110      ' Dim light module
' Create aliases for the pins used
Symbol DataPin = PORTA.1
Symbol ZeroC = PORTA.0
House = 0                  ' Set house to 0 (A)
Unit = 8                   ' Set unit to 8 (9)
' Turn on unit 8 in house 0
Xout DataPin ,ZeroC,[House \ Unit,House \ UnitOn ]
' Turn off all the lights in house 0
Xout DataPin ,ZeroC,[House \ LightsOff ]
Xout DataPin ,ZeroC,[House \ 0]
' Blink light 0 on and off every 10 seconds
Loop:
Xout DataPin ,ZeroC,[House \ UnitOn ]
DelayMs 10000              ' Wait 10 seconds
Xout DataPin ,ZeroC,[House \ UnitOff ]
DelayMs 10000              ' Wait 10 seconds
Goto Loop
```

Xout and Xin Declares

In order to make the **Xout** command's results more in keeping with the BASIC Stamp interpreter, two declares have been included for both Xin and **Xout**. These are.

```
Declare Xout_Translate = On/Off, True/False or 1/0
and
Declare Xin_Translate = On/Off, True/False or 1/0
```

Notes

Xout only transmits data at each zero crossing of the AC power line, as received on **ZeroPin**. If there are no transitions on this line, **Xout** will effectively wait forever.

Xout is used to transmit information from X-10 devices that can receive the appropriate data. X-10 modules are available from a wide variety of sources under several trade names. An interface is required to connect the microcontroller to the AC power line. Either the PL-513 for send only, or the TW-523 for two-way X-10 communications are required. These devices contain the power line interface and isolate the PICmicro™ from the AC line.

The KeyCode numbers and their corresponding operations are listed below: -

KeyCode	KeyCode No.	Operation
UnitOn	%10010	Turn module on
UnitOff	%11010	Turn module off
UnitsOff	%11100	Turn all modules off
LightsOn	%10100	Turn all light modules on
LightsOff	%10000	Turn all light modules off
Bright	%10110	Brighten light module
Dim	%11110	Dim light module

Wiring to the X-10 interfaces requires 4 connections. Output from the X-10 interface (zero crossing and receive data) are open-collector, which is the reason for the pull-up resistors on the microcontroller.

Wiring for each type of interface is shown below: -

PL-513 Wiring

Wire No.	Wire Colour	Connection
1	Black	Zero crossing output
2	Red	Zero crossing common
3	Green	X-10 transmit common
4	Yellow	X-10 transmit input

TW-523 Wiring

Wire No.	Wire Colour	Connection
1	Black	Zero crossing output
2	Red	Common
3	Green	X-10 receive output
4	Yellow	X-10 transmit input

Warning. Under no circumstances should the microcontroller be connected directly to the AC power line. Voltage potentials carried by the power line will not only instantly destroy the microcontroller, but could also pose a serious health hazard.

See also : Xin.

Using the Optimiser

The underlying assembler code produced by the compiler is the single most important element to a good language, because compact assembler not only means more can be squeezed into the tight confines of the microcontroller, but also the code runs faster which allows more complex operations to be performed. This is why the compiler now has a “dead code removal” pass as standard which will remove redundant mnemonics, and replace certain combinations of mnemonics with a single mnemonic. WREG tracking is also implemented as standard which helps eliminate unnecessary loading of a constant value into the WREG.

And even though the compiler already produces good underlying assembler mnemonics, there is always room for improvement, and that improvement is achieved by a separate optimising pass.

The optimiser is enabled by issuing the **Declare**: -

```
Declare Optimiser_Level = n
```

Where n is the level of optimisation required.

The **Declare** should be placed at the top of the BASIC program, but anywhere in the code is actually acceptable because once the optimiser is enabled it cannot be disabled later in the same program.

As of version 3.3.3.0 of the compiler, the optimiser has 3 levels, 4 if you include Off as a level.

Level 0 disables the optimiser.

Level 1 Chooses the appropriate branching mnemonics when using an 18F device, and actively chooses the appropriate page switching mnemonics when using a 14-bit core (16F) device.

This is the single most important optimising pass for larger microcontrollers. For 18F types it will replace **Call** with **RCall** and **Goto** with **Bra** whenever appropriate, saving 1 byte of code space every time.

Level 2 Further re-arranging of branching operations.

Level 3 18F devices only. Re-arranges conditional branching operations. This is an important optimising pass because a single program can implement many decision making mnemonics.

You must be aware that optimising code, especially paged code found in the larger standard 14-bit core (16F) devices can, in some circumstances, have a detrimental effect on a program if it misses a page boundary, this is true of all optimisation on all compilers and is something that you should take into account. This is why the 14-bit core optimiser is not an official part of the compiler, and has been left in place because of current user requests.

Always try to write and test your program without the optimiser pass. Then once it's working as expected, enable the optimiser a level at a time. However, this is not always possible with larger programs that will not fit within the microcontroller without optimisation. In this circumstance, choose level 1 optimisation whenever the code is reaching the limits of the microcontroller, testing the code as you go along.

Caveats

Of course there's no such thing as a free lunch, and there are some features that cannot be used when implementing the optimiser.

The main one is that the optimiser is not supported with 12-bit core devices.

Also, the **Org** directive is not allowed with 14-bit core (16F) devices when using the optimiser, but can be used with 18F devices.

When using 18F devices, do not use the **Movfw** macro as this will cause problems withing the Asm listing, use the correct mnemonic of **Movf** Var, **W**.

On all devices, do not use the assembler **LIST** and **NOLIST** directives, as the optimiser uses these to sculpt the final Asm used.

```
Declare Dead_Code_Remove = On/Off
```

The above declare removes some redundant op-codes from the underlying Asm code.

Removal of redundant Bank Switching mnemonics.

Removal of redundant **Movwf** mnemonics if preceded by a **Movf** Var,W mnemonic.

Removal of redundant **Movf** Var,W mnemonics if preceded by a **Movwf** mnemonic.

Removal of redundant **Andlw** mnemonics if preceded by another **Andlw** mnemonic.

Replaced a **Call-Return** mnemonic pair with a single **Goto** mnemonic.

Note that the Optimiser for standard 14-bit core devices is no longer officially supported, and only remains because of user requests.

Using the Preprocessor

A preprocessor directive is a non executable statement that informs the compiler how to compile. For example, some microcontroller have certain hardware features that others don't. A preprocessor directive can be used to inform the compiler to add or remove source code, based on that particular devices ability to support that hardware.

It's important to note that the preprocessor works with directives on a line by line basis. It is therefore important to ensure that each directive is on a line of its own. Don't place directives and source code on the same line.

It's also important not to mistake the compiler's preprocessor with the assembler's preprocessor. Any directive that starts with a dollar "\$" is the compiler's preprocessor, and any directive that starts with a hash "#" is the assembler's preprocessor. They cannot be mixed, as each has no knowledge of the other.

Preprocessor directives can be nested in the same way as source code statements. For example:

```
$ifdef MyValue
    $if MyValue = 10
        $symbol CodeConst = 10
    $else
        $symbol CodeConst = 0
    $endif
$endif
```

Preprocessor directives are lines included in the code of the program that are not BASIC language statements but directives for the preprocessor itself. The preprocessor is actually a separate entity to the compiler, and, as the name suggests, preprocesses the BASIC code before the actual compiler sees it. Preprocessor directives are always preceded by a dollar sign "\$".

Preprocessor Directives

To define preprocessor macros the directive **\$define** is used. Its format is:-

\$define *identifier replacement*

When the preprocessor encounters this directive, it replaces any occurrence of *identifier* in the rest of the code by *replacement*. This replacement can be an expression, a statement, a block, or simply anything. The preprocessor does not understand BASIC, it simply replaces any occurrence of *identifier* by *replacement*.

```
$define TableSize 100
Dim Table1[TableSize] as Byte
Dim Table2[TableSize] as Byte
```

After the preprocessor has replaced TableSize, the code becomes equivalent to:-

```
Dim Table1[100] as Byte
Dim Table2[100] as Byte
```

The use of **\$define** as a constant definer is only one aspect of the preprocessor, and **\$define** can also work with parameters to define psuedo function macros. The syntax then is:-

\$define *identifier (parameter list) replacement*

A simple example of a function-like macro is:-

```
$define RadToDeg(x) ((x) * 57.29578)
```

This defines a radians to degrees conversion which can be used as:-

```
Var1 = RadToDeg(34)
```

This is expanded in-place, so the caller does not need to clutter copies of the multiplication constant throughtout the code.

Precedence

Note that the example macro RadToDeg(x) given above uses normally unnecessary parentheses both around the argument and around the entire expression. Omitting either of these can lead to unexpected results. For example:-

Macro defined as:

```
$define RadToDeg(x) (x * 57.29578)
```

will expand

```
RadToDeg(a + b)
```

to

```
(a + b * 57.29578)
```

Macro defined as

```
$define RadToDeg(x) (x) * 57.29578
```

will expand

```
1 / RadToDeg(a)
```

to

```
1 / (a) * 57.29578
```

neither of which give the intended result.

Not all replacement tokens can be passed back to an assignment using the equals operator. If this is the case, the code needs to be similar to BASIC Stamp syntax, where the assignment variable is the last parameter:-

```
$define GetMax(x,y,z) If x > y Then z = x : Else : z = y
```

This would replace any occurrence of GetMax followed by three parameter (argument) by the replacement expression, but also replacing each parameter by its identifier, exactly as would be expected of a function.

```
Dim Var1 as Byte  
Dim Var2 as Byte  
Dim Var3 as Byte
```

```
Var1 = 100  
Var2 = 99  
GetMax(Var1, Var2, Var3)
```

The previous would be placed within the BASIC program as:-

```
Dim Var1 as Byte
Dim Var2 as Byte
Dim Var3 as Byte

Var1 = 100
Var2 = 99
If Var1 > Var2 Then Var3 = Var1 : Else : Var3 = Var2
```

Notice that the third parameter “Var3” is loaded with the result.

A macro lasts until it is undefined with the **\$undef** preprocessor directive:-

```
$define TableSize 100
Dim Table1[TableSize] as Byte
$undef TableSize
$define TableSize 200
Dim Table2[TableSize] as Byte
```

This would generate the same code as:-

```
Dim Table1[100] as Byte
Dim Table2[200] as Byte
```

Because preprocessor replacements happen before any BASIC syntax check, macro definitions can be a tricky feature, so be careful. Code that relies heavily on complicated macros may be difficult to understand, since the syntax they expect is, on many occasions, different from the regular expressions programmers expect in Proton BASIC.

Preprocessor directives only extend across a single line of code. As soon as a newline character is found (end of line), the preprocessor directive is considered to end. The only way a preprocessor directive can extend through more than one line is by preceding the newline character at the end of the line by a comment character (') followed by a new line. No comment text can follow the comment character. For example:-

```
$define GetMax(x,y,z) If x > y Then '
                        z = x      '
                        Else      '
                        z = y      '
                        EndIf
```

```
GetMax(Var1, Var2, Var3)
```

The compiler will see:-

```
If Var1 > Var2 Then
    Var3 = Var1
Else
    Var3 = Var2
EndIf
```

Note that parenthesis is always required around the **\$define** declaration and its use within the program.

If the *replacement* argument is not included within the **\$define** directive, the *identifier* argument will output nothing. However, it can be used as an identifier for conditional code:-

```
$define DoThis  
  
$ifdef DoThis  
    {Rest of Code here}  
$endif
```

\$undef *identifier*

This removes any existing definition of the user macro *identifier*.

\$eval *expression*

In normal operation, the **\$define** directive simply replaces text, however, using the **\$eval** directive allows constant value expressions to be evaluated before replacement within the BASIC code. For example:-

```
$define Expression(Prm1) $eval Prm1 << 1
```

The above will evaluate the constant parameter Prm1, shifting it left one position.

```
Var1 = Expression(1)
```

Will be added to the BASIC code as:-

```
Var1 = 2
```

Because 1 shifted left one position is 2.

Several operators are available for use with an expression. These are +, -, *, /, ~, <<, >>, =, >, <, >=, <=, <>, And, Or, Xor.

Conditional Directives (**\$ifdef**, **\$ifndef**, **\$if**, **\$endif**, **\$else** and **\$elseif**)

Conditional directives allow parts of the code to be included or discarded if a certain condition is met.

\$ifdef allows a section of a program to be compiled only if the macro that is specified as the parameter has been defined, no matter what its value is. For example:-

```
$ifdef TableSize  
    Dim Table[TableSize] as Byte  
$endif
```

In the above condition, the line of code *Dim Table[TableSize] as Byte* is only compiled if TableSize was previously defined with **\$define**, independent of its value. If it was not defined, the line will not be included in the program compilation.

\$ifndef serves for the exact opposite: the code between **\$ifndef** and **\$endif** directives is only compiled if the specified identifier has not been previously defined. For example:-

```
$ifndef TableSize  
    $define TableSize 100  
$endif  
Dim Table[TableSize] as Byte
```

In the previous code, when arriving at this piece of code, the `TableSize` directive has not been defined yet. If it already existed it would keep its previous value since the `$define` directive would not be executed.

A valuable use for `$ifdef` is that of a code guard with include files. This allows multiple insertions of a file, but only the first will be used.

A typical code guard looks like:

```
$ifndef IncludeFileName
    $define IncludeFileName
    { BASIC Code goes Here }
$endif
```

The logic of the above snippet is that if the include file has not previously been loaded into the program, the `$define IncludeFileName` will not have been created, thus allowing the inclusion of the code between `$ifndef` and `$endif`. However, if the include file has been previously loaded, the `$define` will have already been created, and the condition will be false, thus not allowing the code to be used.

`IncludeFileName` must be unique to each file. Therefore, it is recommended that a derivative of the Include File's name is used.

`$if expression`

This directive invokes the arithmetic evaluator and compares the result in order to begin a conditional block. In particular, note that the logical value of *expression* is always true when it cannot be evaluated to a number.

The `$if` directive as well as the `$elseif` directive can use quite complex logic. For example:-

```
$if _device = _18F452 or _device = _18F4520 and _core = 16
    { BASIC Code Here }
$endif
```

There are several built in user defines that will help separate blocks of code. These are:-

- `_device`. This holds the PICmicro™ device name, as a string. i.e. `_18F452`, `_12F508`, `_16F684` etc. Notice the preceding underscore
- `_core`. This holds the device's core. i.e. 12 for 12-bit core devices, 14 for 14-bit core (16F) devices, and 16 for 18F devices.
- `_ram`. This holds the amount of RAM contained in the device (in bytes).
- `_code`. This holds the amount of flash memory in the device. In *words* for 12 and 14-bit core devices, and *bytes* for 18F devices.
- `_eeprom`. This holds the amount of eeprom memory the device contains.
- `_ports`. This holds the amount of I/O ports that the device has.
- `_adc`. This holds the amount of ADC channels the device has.
- `_adcsres`. This holds the resolution of the device's ADC. i.e. 8, 10, or 12.
- `_usart`. This holds the amount of USARTS the device has. i.e. 0, 1, or 2
- `_usb`. This holds the amount of USB peripherals the device has. i.e. 0 or 1
- `_flash`. This informs of the ability for the device to access it's own code memory. 0 = no access, 1 = read and write, and 2 = read only

The values for the user defines are taken from the compiler's PPI files, and are only available if the compiler's **Device** directive is included within the BASIC program.

\$else

This toggles the logical value of the current conditional block. What follows is evaluated if and only if the preceding input was commented out.

\$endif

This ends a conditional block started by the **\$if** directive.

\$elseif *expression*

This directive can be used to avoid nested **\$if** conditions. **\$if..\$elseif..\$endif** is equivalent to **\$if..\$else \$if ..\$endif \$endif**.

The **\$if**, **\$else** and **\$elseif** directives serve to specify some condition to be met in order for the portion of code they surround to be compiled. The condition that follows **\$if** or **\$elseif** can only evaluate constant expressions, including macro expressions. For example:-

```
$if TableSize > 200
    $undef TableSize
    $define TableSize 200
```

```
$elseif TableSize < 50
    $undef TableSize
    $define TableSize 50
```

```
$else
    $undef TableSize
    $define TableSize 100
$endif
```

```
Dim Table[TableSize] as Byte
```

Notice how the whole structure of **\$if**, **\$elseif** and **\$else** chained directives ends with **\$endif**.

The behavior of **\$ifdef** and **\$ifndef** can also be achieved by using the special built-in user directive **_defined** and **!_defined** respectively, in any **\$if** or **\$elseif** condition. These allow more flexibility than **\$ifdef** and **\$ifndef**. For example:-

```
$if _defined (MyDefine) and _defined (AnotherDefine)
    { BASIC Code Here }
$endif
```

The argument for the **_defined** user directive must be surrounded by parenthesis. The preceding character "!" means "not".

\$error *message*

This directive causes an error message with the current filename and line number. Subsequent processing of the code is then aborted.

```
$error Error Message Here
```

Using the Proton Compiler with MPLAB IDE™

The compiler can be used within the MPLAB™ IDE environment and allows single stepping of the code on a high-level basis. i.e. BASIC lines of code, or the use of the ICD2, PICKit 2, PICKit3, or a device Programmer.

We'll walk through the method of operation step by step.

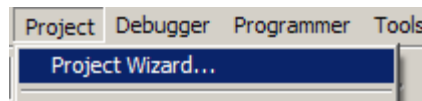
First, download a copy of the latest MPLAB™ IDE because this method will only work on versions 8.0 onwards. The release at the time of writing is 8.60, and it is recommend to use this version. MPLAB™ can be downloaded from www.microchip.com

Locate the files *tlchill.ini* and *proton.mtc* within the compiler's folder (default location *C:\Program Files\ProtonIDE\PDS*) and copy them into MPLAB's folder "Core\MTC Suites", overwriting any previous files. MPLAB™ will default to location *C:\Program Files\Microchip\MPLAB IDE*, therefore, the legacy folder should be located at:

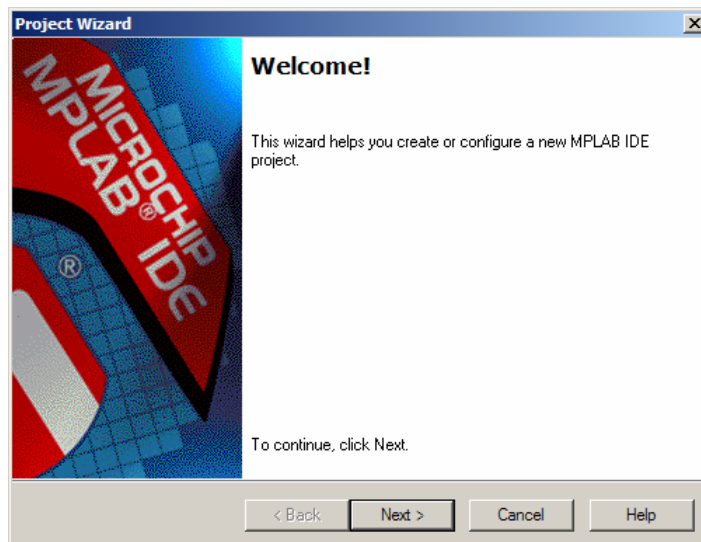
C:\Program Files\Microchip\MPLAB IDE\Core\MTC Suites.

Once these files have been copied, locate and run the file *Mplab_Proton.reg*, which can also be found within the compiler's folder. This will add entries into the registry that will register the Proton Compiler as a toolsuite within MPLAB™. If using a 64-bit OS, run the file *Mplab_Proton_64.reg* instead.

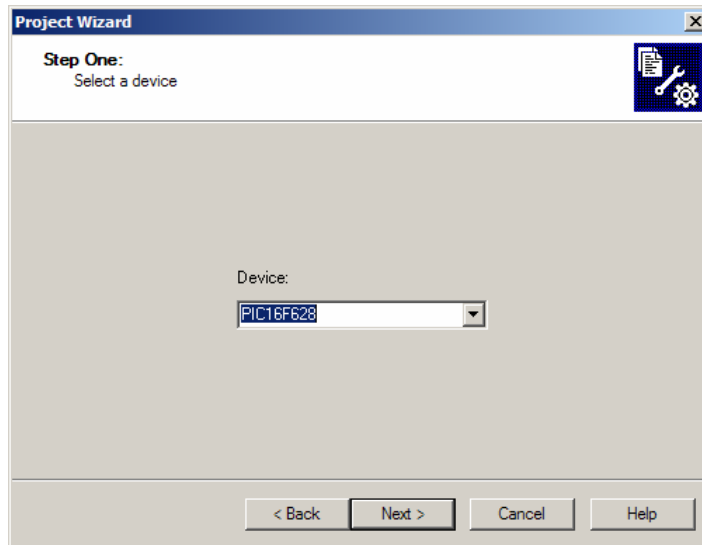
Open MPLAB™, then click on the *Project Wizard* menu option.



This will open the intro window as shown below.



For this demonstration program, the microcontroller of choice is the 16F628A, so in the step 1 window, choose the 16F628A device.

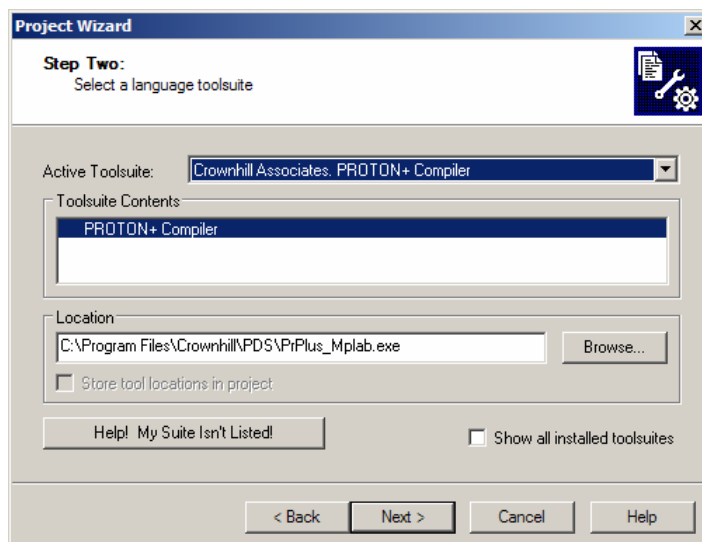


By default the device chosen in this window will be the device that the compiler uses, regardless of a **Device** directive within the BASIC listing. The **Device** directive will be ignored (see end of document to disable this).

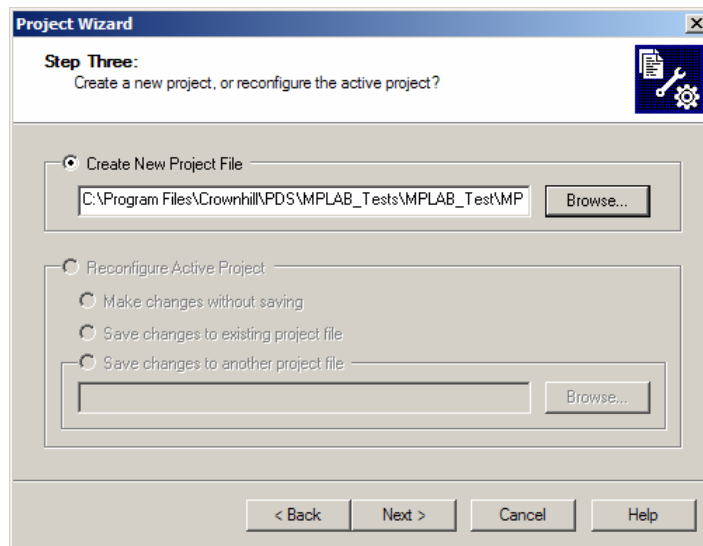
Click *Next*, then choose the Crownhill Associates Proton Compiler toolsuite, and *browse* to where the Proton compiler's executable is stored.

The default location for this is *C:\Program Files\ProtonIDE\PDS*.

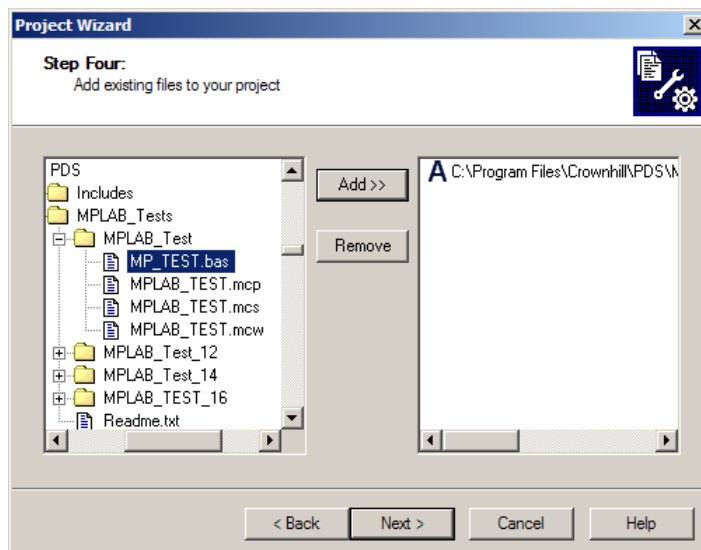
Browse to the file named **PrPlus.exe** and enter this in the Location window. It should be within the compiler's folder.



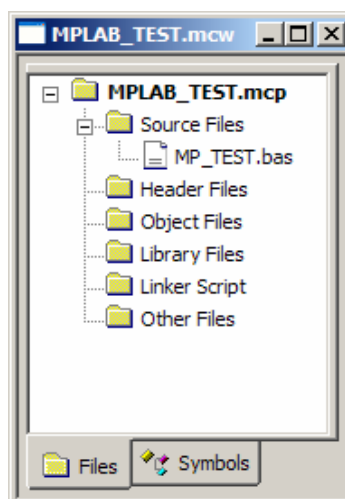
After clicking *Next*, a project name and location needs to be chosen in the step 3 window. The name given to the demonstration project is MPLAB_Test, and it's located, in this case, in the compiler's source code folder. But it can be placed virtually anywhere on the hard drive as long as it is not nested too deeply.



Now we need to add the BASIC file to the project. The BASIC file for the demonstration is named MP_Test.bas.

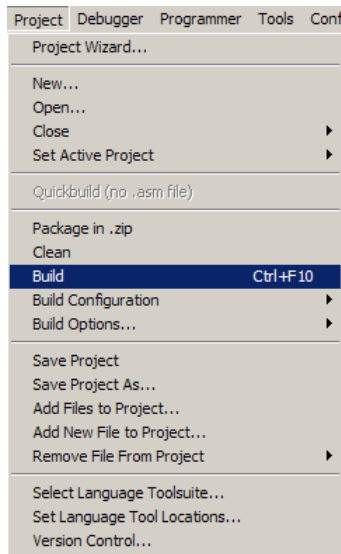


Clicking *Next* a few times after step 4 will create the project. But no BASIC filename has been loaded into the IDE, so right click on *Source File* option located in the *MPLAB_Test.mcw* window, and choose the appropriate BASIC file.

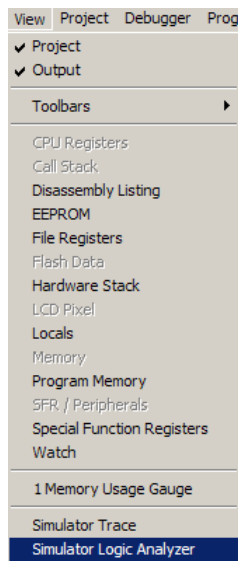


8-bit Proton Compiler Development Suite.

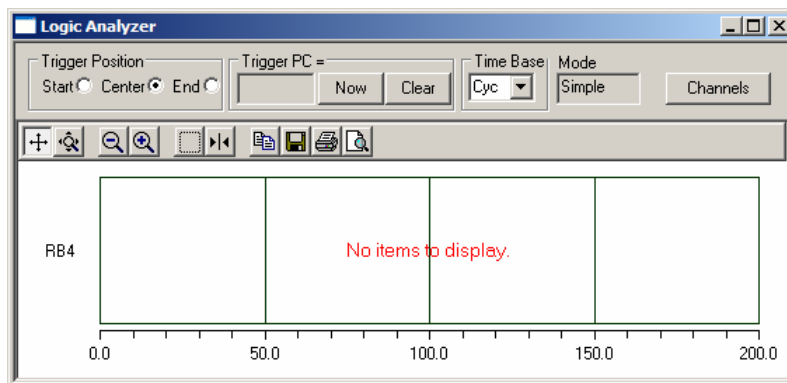
Double click on the *MP_Test.bas* text in the *MPLAB_Text.mcw* window, and the BASIC file will be opened ready to compile. Choose *Project* then *Build* or (*Ctrl F10*) to compile the program.



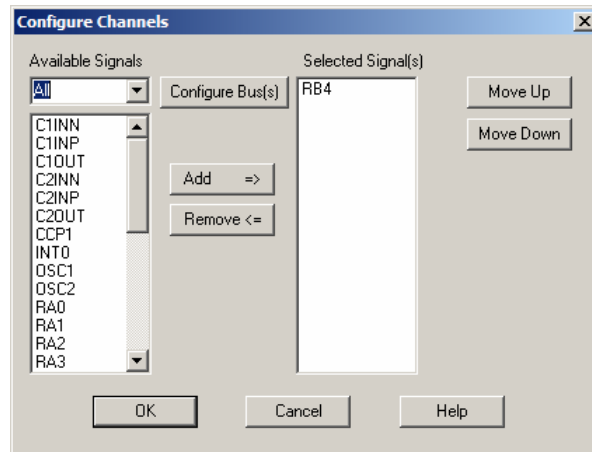
For this demonstration the Logic Analyser will be used. Choose *View->Simulator Logic Analyser* from the menu bar.



The logic analyser window will then appear.



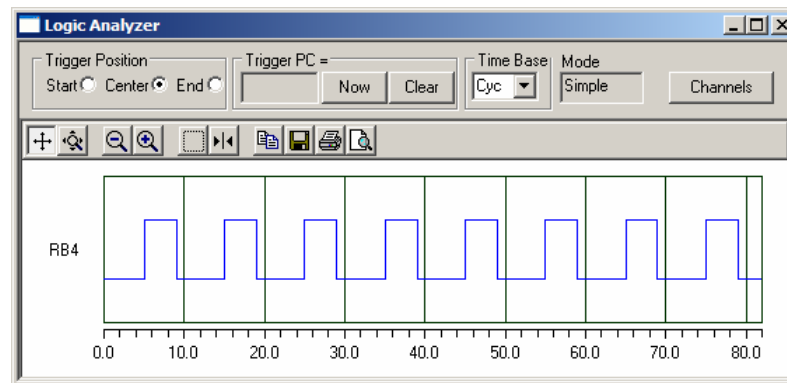
Click on the Channels button and choose the Port and Pin used in the BASIC program. i.e. PORTB.4.



Run the simulator using the animate icon.

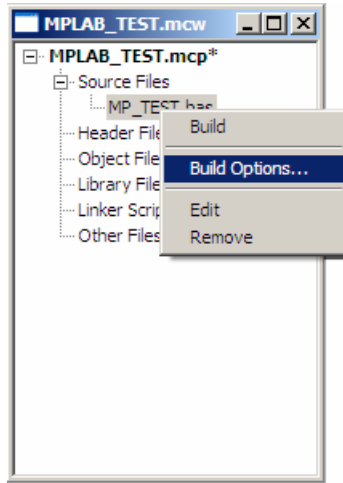


And the logic analyser will show the condition of PORTB.4.

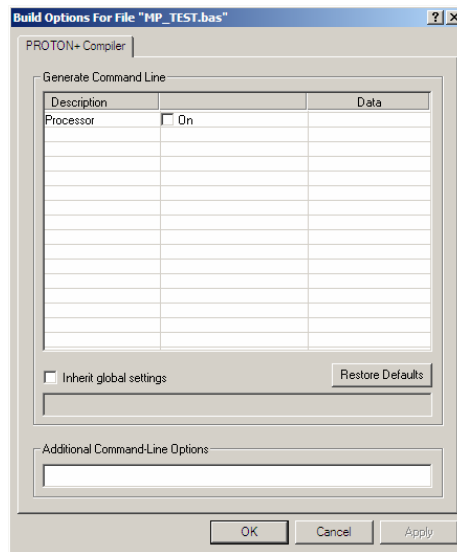


Disabling the Automatic Device Selection.

By default, MPLAB™ forces the compiler to ignore any **Device** directives within the BASIC program in favour of whatever device is chosen in the *Configure->Select Device Options* menu. This can be disabled by right clicking on the filename within the *MCW* window and choosing *Build Options*.



You will be presented with a configuration window containing a single switch.



Untick the *On* switch and click *Apply*.

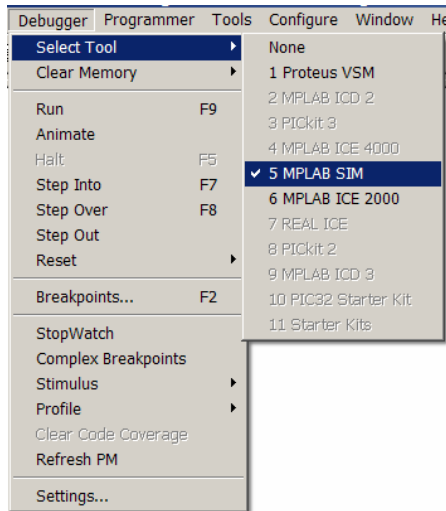
The microcontroller that the compiler recognises is now issued by the **Device** directive within the BASIC program, therefore ensure that MPLAB™ is configured for the correct device for any simulations of programming.

An Easier Method

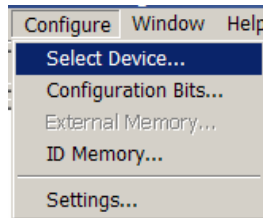
There is another way to simulate within MPLAB™ without going through the tedious process of creating a project.

When the directive `Declare Create_Coff = On` is placed within the BASIC program, a cof file (common Object File) is produced during compilation. A cof file has all the information required for simulation, and is as close as it gets to a standard format.

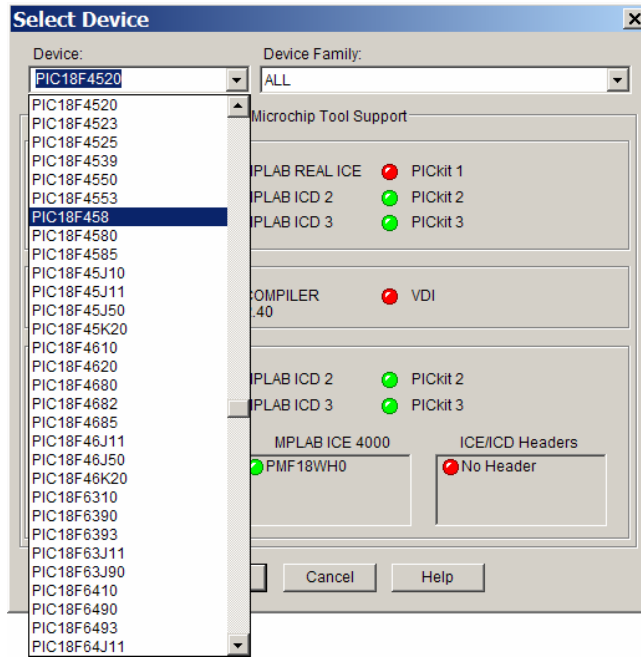
Open MPLAB™, and close any open projects, this is an important procedure. You should now be presented with an empty workspace. Choose the debugger of choice from the *debugger* toolbar menu.



Then choose the appropriate device that the BASIC program is compiled for, by clicking on the *configure->select device* toolbar menu.



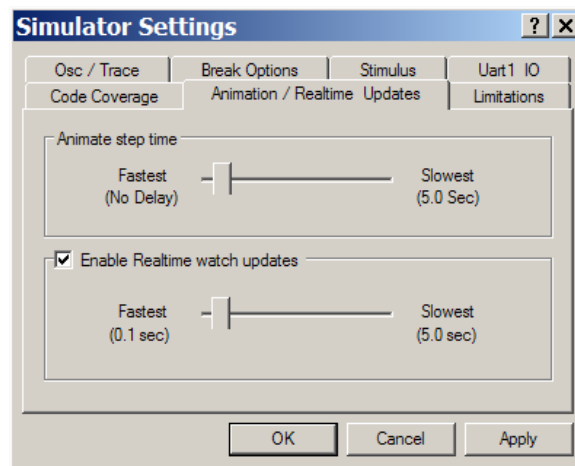
You can now choose the device from a list.



Open the folder where the BASIC file was situated, and drag the file with the extension '.cof' on to the the MPLAB™ workspace. It will be automatically opened to show the BASIC file.

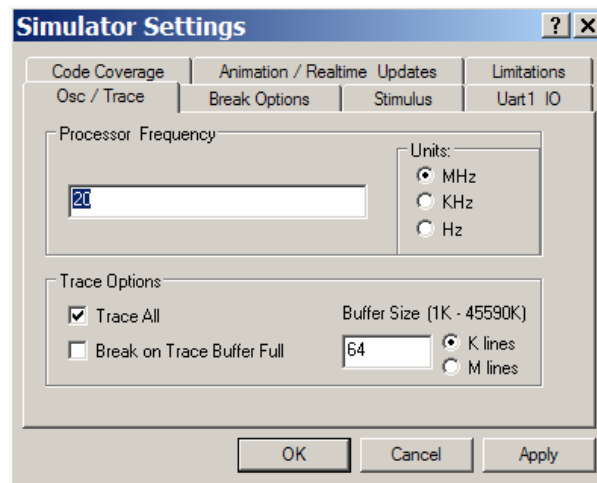
The program can now be simulated, either by animation or single stepping. However, there are still 2 steps to carry out that will improve the simulation. Click on the *debugger* toolbar menu, and choose the bottom option. i.e. *settings*.

When the window shows, click the *Animation/Realtime Updates* tab.



Tick *Enable Realtime watch updates*, and move the step time closer to the fastest side. i.e. far left.

Now click on the *Osc/Trace* tab, and choose the oscillator frequency used in the BASIC program.



Click *Apply* for the settings to take effect, then *OK*.

You can now open watch windows, disassembly listings etc, and watch the variables update as the simulation is in progress.

Protected Compiler Words

Below is a list of protected words that the compiler or assembler uses internally. Be sure not to use any of these words as variable or label names, otherwise errors will be produced.

(A)

Abs, Access_Upper_64K, Acos, Actual_Banks, ADC_Resolution, Adcin, Addlw, Addwfc, Addwfc, Adin, Adin_Delay, Adin_Res, Adin_Stime, Adin_Tad, All_Digital, Andlw, Asin, Asm, Atan, Auto_Context_Save, Available_RAM

(B)

Bank0_End, Bank0_Start, Bank10_End, Bank10_Start, Bank11_End, Bank11_Start, Bank12_End, Bank12_Start, Bank13_End, Bank13_Start, Bank14_End, Bank14_Start, Bank15_End, Bank15_Start, Bank1_End, Bank1_Start, Bank2_End, Bank2_Start, Bank3_End, Bank3_Start, Bank4_End, Bank4_Start, Bank5_End, Bank5_Start, Bank6_End, Bank6_Start, Bank7_End, Bank7_Start, Bank8_End, Bank8_Start, Bank9_End, Bank9_Start, Bank_Select_Switch, BankSel, BankSel, Bc, Bcf, Bin, Bin1, Bin10, Bin11, Bin12, Bin13, Bin14, Bin15, Bin16, Bin17, Bin18, Bin19, Bin2, Bin20, Bin21, Bin22, Bin23, Bin24, Bin25, Bin26, Bin27, Bin28, Bin29, Bin3, Bin30, Bin31, Bin32, Bin4, Bin5, Bin6, Bin7, Bin8, Bin9, Bit, Bn, Bnc, Bnn, Bnov, Bnz, Bootloader, Bov, Box, Bra, Branch, Branchl, Break, Brestart, Bsf, Bstart, Bstop, Btfsc, Btfss, Btg, Bus_DelayMs, Bus_SCL, BusAck, Busin, Busout, Button, Button_Delay, Byte, Byte_Math, BZ, Bit_Bit, Bit_Byte, Bit_Dword, Bit_Float, Bit_Word, Bit_Wreg, Byte_Bit, Byte_Byte, Byte_Dword, Byte_Float, Byte_Word, Byte_Wreg, Brw

(C)

Call, Case, Cblock, CCP1_Pin, CCP2_Pin, CCP3_Pin, CCP4_Pin, CCP5_Pin, Cdata, Cerase, CF_ADPort, CF_ADPort_Mask, CF_CD1Pin, CF_CE1Pin, CF_DTPort, CF_Init, CF_OEPin, CF_RDYPin, CF_Read, CF_Read_Write_Inline, CF_RSTPin, CF_Sector, CF_WEPin, CF_Write, Chr\$, Circle, Clear, ClearBit, Clrf, Clrw, Cls, Code, Comf, Config, Constant, Context, Continue, Core, Cos, Count, Counter, Cpfseq, Cpfsgt, Cpflst, Cread, Cread8, Cread16, Cread32, Cursor, Cwrite, Callw, cPtr8, cPtr16, cPtr32

(D)

Da, Data, Daw, Db, Dc, Dcd, Dcfsnz, De, Dead_Code_Remove, Dword_Bit, Dword_Byte, Dword_Dword, Dword_Float, Dword_Word, Dword_Wreg, Debug_Req, Debugin, Dec, Dec, Dec1, Dec1, Dec10, Dec2, Dec2, Dec3, Dec3, Dec4, Dec4, Dec5, Dec5, Dec6, Dec6, Dec7, Dec7, Dec8, Dec8, Dec9, Decf, Decfsz, Declare, Dctrment, Define, Delayms, Delayus, DelayCs, Device, Dig, Dim, Disable, Div32, Djc, Djnc, Djnz, Djz, Dt, DTMfout, Dw, Dword

(E)

Edata, Eeprom_Size, Else, Elseif, Enable, End, EndAsm, Endlf, EndM, EndSelect, equ, Eread, Error, ErrorLevel, Ewrite, ExitM, Exp, Expand

(F)

Fill, Fix16_8Add, Fix16_8Div, Fix16_8Greater, Fix16_8GreaterEqual, Fix16_8Less, Fix16_8LessEqual, Fix16_8Mul, Fix16_8Sub, Fix16_8ToFloat, Fix16_8ToInt, Fix8_8Add, Fix8_8Div, Fix8_8Greater, Fix8_8GreaterEqual, Fix8_8Less, Fix8_8LessEqual, Fix8_8Mul, Fix8_8Sub, Fix8_8ToFloat, Fix8_8ToInt, Flash_Capable, Float, Float_Display_Type, Float_Rounding, FloatToFix16_8, FloatToFix8_8, Font_Addr, For, Freqout, Float_Bit, Float_Byte, Float_Dword, Float_Float, Float_Word, Float_Wreg

(G)

Get#Proton#Version, GetBit, GLCD_CS_Invert, GLCD_Fast_Strobe, GLCD_Read_Delay, GLCD_Strobe_Delay, Gosub, Goto

(H)

HbRestart, HbStart, HbStop, Hbus_Bitrate, HbusAck, Hbusin, Hbusout, Hex, Hex1, Hex2, Hex3, Hex4, Hex4, Hex5, Hex6, Hex7, Hex8, High, High_Int_Sub_End, High_Int_Sub_Start, HighLow_Tris_Reverse, Hpwm, Hrsin, Hrsin2, Hrsout, Hrsout2, Hserial2_Baud, Hserial2_Clear, Hserial2_Parity, Hserial2_RCSTA, Hserial2_SPBRG, Hserial2_TXSTA, Hserial_Baud, Hserial_Clear, Hserial_Parity, Hserial_RCSTA, Hserial_SPBRG, Hserial_TXSTA, Hserin, Hserin2, Hserout, Hserout2

(I)

I2C_Bus_SCL, I2C_Slow_Bus, I2Cin, I2Cout, I2CWrite, I2CRead, ICD_Req, Icos, Idata, If, Ijc, Ijnc, Ijnz, Ijz, Inc, Incf, Incfsz, Include, Increment, Infsnz, Inkey, Input, Int_Sub_End, Int_Sub_Start, Internal_Bus, Internal_Font, IntToFix16_8, IntToFix8_8, Iorlw, Iorwf, Irln, Irln_Pin, Isin, ISqr

(K)

Keyboard_CLK_Pin, Keyboard_DTA_Pin, Keyboard_IN, Keypad_Port

(L)

Label_Word, Label_Bank_Resets, LCD_CDPin, LCD_CEPin, LCD_CommandUS, LCD_CS1Pin, LCD_CS2Pin, LCD_DataUs, LCD_DTPin, LCD_DTPort, LCD_ENPin, LCD_Font_HEIGHT, LCD_Font_Width, LCD_Graphic_Pages, LCD_Interface, LCD_Lines, LCD_RAM_Size, LCD_RDPin, LCD_RSPin, LCD_RSTPin, LCD_RWPin, LCD_Text_Home_Address, LCD_Text_Pages, LCD_Type, LCD_WRPin, LCD_X_Res, LCD_Y_Res, LCDread, LCDwrite, Ldata, Left\$, Len, Let, Lfsr, Lslf, Lsrf, Library_Core, Line, LineTo, LoadBit, Local, Log, Log10, LookDown, LookDownL, LookUp, LookUpL, Low, Low_Int_Sub_End, Low_Int_Sub_Start, Lread, Lread16, Lread32, Lread8

(M)

Macro_Params, Max, Mid\$, Min, Mouse_CLK_Pin, Mouse_Data_Pin, Mouse_In, Movf, Movff, Movlw, Movwf, Mssp_Type, Mullw, Mulwf, Movwi, Moviw

(N)

Ncd, Negf, Next, Nop, Num_Bit, Num_Byte, Num_Dword, Num_Float, Num_FSR, Num_FSR0, Num_FSR2, Num_Word, Num_Wreg

(O)

On_Hard_Interrupt, On_Hardware_Interrupt, On_Interrupt, On_Low_Interrupt, On_Soft_Interrupt, On_Software_Interrupt, Onboard_Adc, Onboard_Uart, Onboard_Usb, Optimise_Bit_Test, Optimiser_Level, Oread, Org, Output, Owin, Owout, Owrite

(P)

Page, PageSel, Pause, Pauseus, Peek, PIC_Pages, Pixel, PLL_Req, Plot, Poke, Pop, Portb_Pullups, Pot, Pow, Print, Prm_1, Prm_10, Prm_11, Prm_12, Prm_13, Prm_14, Prm_15, Prm_2, Prm_3, Prm_4, Prm_5, Prm_6, Prm_7, Prm_8, Prm_9, Prm_Count, Proton_Start_Address, PulsIn, PulseIn, Pulsin_Maximum, PulseOut, Push, Pwm, Ptr8, Ptr16, Ptr32

(R)

RAM_Bank, RAM_Banks, Random, RC5in, RC5in_Extended, RC5in_Pin, RCall, RCin, RcTime, Read, Rem, Remarks, Reminders, Rep, Repeat, Res, Reserve_RAM, Reset_Bank, Restore, Resume, Retfie, Retlw, Return, Return_Type, Return_Var, Rev, Right\$, Rlcf, Rlcf, Rlncf, Rol, Ror, Rrcf, Rrf, Rrncf, Rsin, Rsin_Mode, Rsin_Pin, Rsin_Timeout, Rsout, Rsout_Baud,

(R continued)

Rsout_Mode, Rsout_Pace, Rsout_Pin, Return_Bit, Return_Byte,
Return_Dword, Return_Float, Return_Word, Return_Wreg

(S)

SBreak, SCL_Pin, SDA_Pin, Seed, Select, Serial_Baud, Serial_Data,
Serial_Parity, Serin, Serout, Servo, Set, Set_Bank, Set_Defaults, Set_OSCCAL, SetBit, Setf,
Shift_DelayUs, ShiftIn, Shin, Shout, Show_Expression_Parts, Show_System_Variables,
Signed_Dword_Terms, Sin, Single_Page_Model, SizeOf, Sleep, Slow_Bus,
Small_Micro_Model, Snooze, SonyIn, SonyIn_Pin, Sound, Sound2, Sqr, Stack_Size, Stamp
_Cos, Stamp_Sin, Stamp_Sqr, Step, Stop, Str, Str\$, Str\$, StrCmp, String, Strn, Subfwb,
Sublw, Subwf, Subwfb, Swap, Swapf, Symbol

(T)

Tan, Tblrd, Tblwt, TCase, Then, to, Toggle, ToLower, Toshiba_Command, Toshiba_UDG,
ToUpper, Tstfsz

(U)

Udata, UnPlot, Unsigned_Dwords, Until, Upper, USB_At_TOM, USB_Class_File,
USB_Count_Errors, USB_Descriptor,
USB_Self_Power_Pin, USB_Sense_Pin, USB_Show_Enum, USB_Type, USBin,
USBin_Auto_Poll, USBin_Buffer_Length, USBin_Buffer_Start, USbinit,
USBout, USBout_Auto_Poll, USBout_Buffer_Length, USBout_Buffer_Start, USBpoll,
USBService

(V)

Val, Var, Variable, VarPtr

(W)

Wait, Warnings, WatchDog, Wend, While, Word, Write, Word_Bit, Word_Byte, Word_Dword,
Word_Float, Word_Word, Word_Wreg, Wreg_Bit, Wreg_Byte, Wreg_Dword, Wreg_Float,
Wreg_Word

(X)

Xin, Xorlw, Xorwf, Xout, Xtal

_adc, _adcsres, _code, _core, _defined, _device, _eeprom, _flash, _mssp, _ports, _ram, _uart
_usb, _xtal

Notes.