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

Web URLs correct at time of publication.

The Proton24 compiler and documentation is written and maintained 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, 2013.

Introduction

The Proton24 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 microcontroller without having to learn the relative complexity of assembler, or wade through the potential gibberish that is C.

The Proton IDE provides a seamless development environment, which allows you to write 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.

The Proton24 compiler allows four devices without requiring a USB key. The supported free devices are: 24EP128MC202, 24FJ64GA002, 24FJ64GA004, and 24HJ128GP502.

Contact Details

For your convenience we have set up a web site **www.protonbasic.co.uk**, where there is a section for users of the Proton24 compiler, to discuss the compiler, and provide self help with programs written for Proton24 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,
Wilburton,
Ely,
Cambridgeshire.
CB6 3PZ.

Telephone

(+44) 01353 749990

Fax

(+44) 01353 749991

Email

sales@crownhill.co.uk

Web Sites

http://www.crownhill.co.uk
http://www.protonbasic.co.uk

Table of Contents.

Proton IDE Overview

The Proton IDE is a professional and powerful Integrated Development Environment (IDE) designed specifically for the Proton24 compiler. It is designed to accelerate product development in a comfortable user friendly environment without compromising performance, flexibility or control.

Code Explorer
Allows quick navigation through the program code and device SFRs (Special Function Registers) .

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 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).

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, Baud rate, parity, byte size and number of stop bits. Alternatively, you can use Serial Communicator favourites to quickly load pre-configured connection settings.

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

Supported Operating Systems
Windows 7 32-bit or 64-bit

Hardware Requirements
At least a 1 GHz Processor
At least 1 GB of RAM
At least 40 GB of hard drive space for the O/S and applications etc.

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.

- **Online Updates** - Executes the IDE online update process, which checks online and installs the latest IDE updates.

Help Menu

- **Help Topics** - Displays the help file section for the toolbar.

- **Online Forum** - Opens your default web browser and connects to the online Proton24 Plus developer forum.

- **About** - Display about dialog, giving both the Proton IDE and Proton24 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.

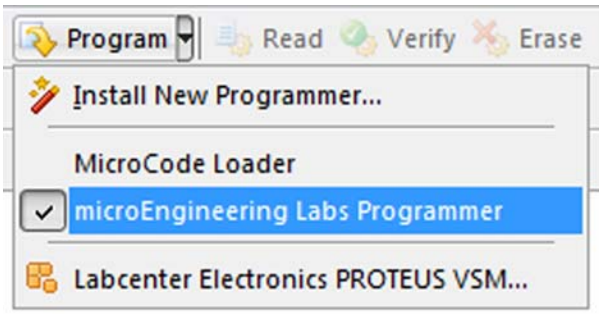
Block Uncomment
Removes the comment character from each line of a selected block of text. If multiple lines are not selected, a single comment is removed from the start of the line containing the cursor.

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. This enables you to program the generated *.hex file into the microcontroller. Alternatively, the compiler output can be sent to an IDE Plugin. 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, melab's USB programmer 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.

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

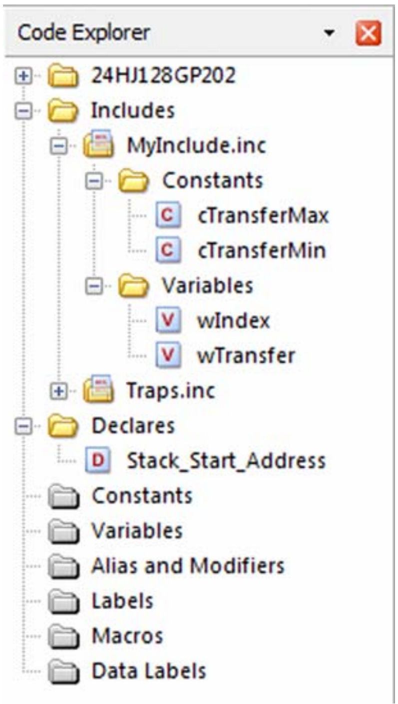
The device node is the first node in the explorer tree. It displays your currently selected processor type. For example, if you program has the declaration: -

```
Device = 24HJ128GP502
```

then the name of the device node will be 24HJ128GP502. 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 explorer 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.inc has expanded the node to reveal its contents. It can now be see that MyInclude.inc has two constant declarations called cTransferMax and cTransferMin and also two variables called wIndex and wTransfer. The include file also contains another include file called Traps.inc. Again, by clicking the icon, the contents of the Traps.inc file can be seen, without opening the file itself. 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 cTransferMax, the include file MyInclude.inc would be opened and the declaration cTransferMax 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.

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 RAM used, the version number of the project and also date and time. Note that RAM usage also includes the microcontroller's stack size.

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 bytes and the number of data bytes used will still be displayed in the IDE status bar.

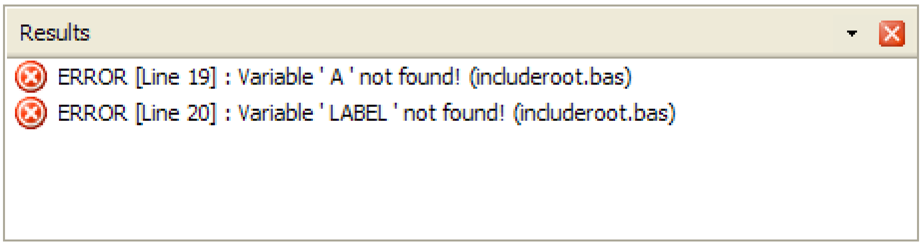
Version Numbers
The version number is automatically incremented after a successful build. Version numbers are displayed as major, minor, release and build. Each number will rollover if it reaches 256. For example, if your version number is 1.0.0.255 and you compile again, the number displayed will be 1.0.1.0. You might want to start you 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 or reminder 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, the 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. The 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.

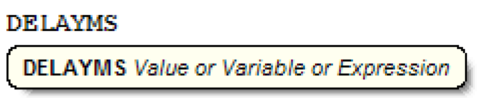
Show Right Gutter
Displays a line to the right of the main editor. You can also set the distance from the left margin (in characters). This feature can be useful for aligning your program comments.

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,



Parameter hints are automatically hidden when the first parameter character is typed. To view the hint again, press F1.

Open Last File(s) When Application Starts
When checked, the documents that were open when the Proton IDE was closed are automatically loaded again when the application is restarted.

Display Full Filename Path in Application Title Bar
By default, The IDE only displays the document filename in the main application title bar (that is, no path information is includes). Check display full pathname if you would like to display additional path information in the main title bar.

Prompt if File Reload Needed

The 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, The 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 Word
```

and you type 'myindex' in the editor window, The 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

The 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 (Proton24)
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 The 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 word-pad 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 Micro-soft Word).

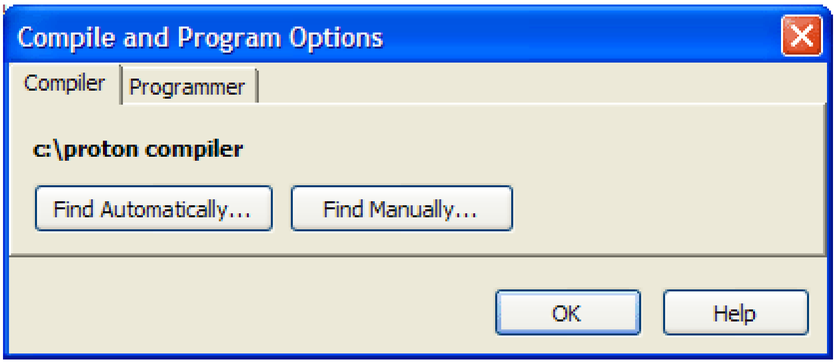
Header options allows you to change the author and copyright name that is placed in a header when a new document is created. For example: -

```
'****************************************************************
'* Name    : Untitled.bas                                       *
'* Author  : J.R Hartley                                        *
'* Notice  : Copyright (c) 2013 MyCompany                       *
'*         : All Rights Reserved                                *
'* Date    : 19/01/13                                           *
'* Version : 1.0                                                *
'* Notes   :                                                    *
'*         :                                                    *
'****************************************************************
```

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

Compile and Program Options

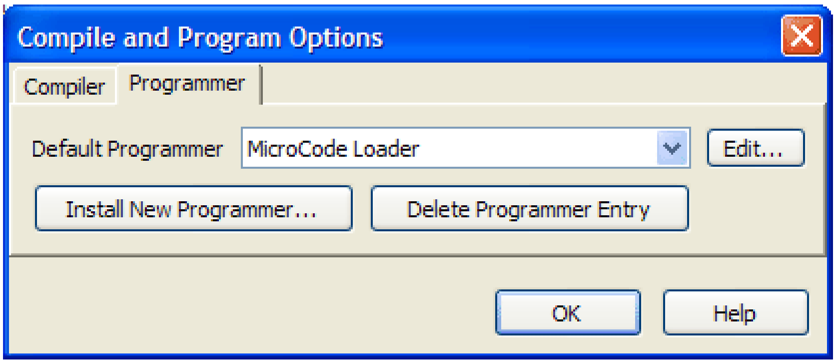
Compiler Tab



You can get the 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.

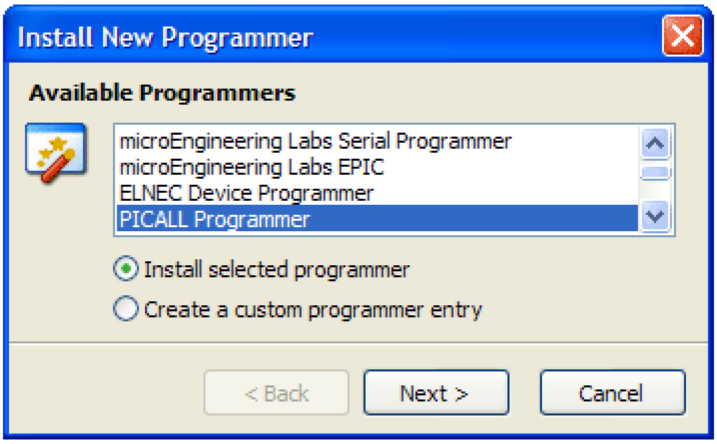
Programmer Tab



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.

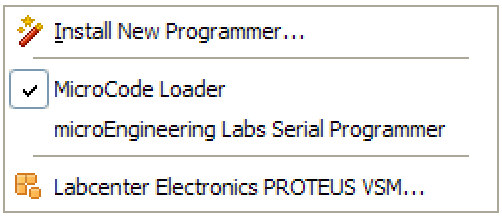
Installing a Programmer

The 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 the 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.



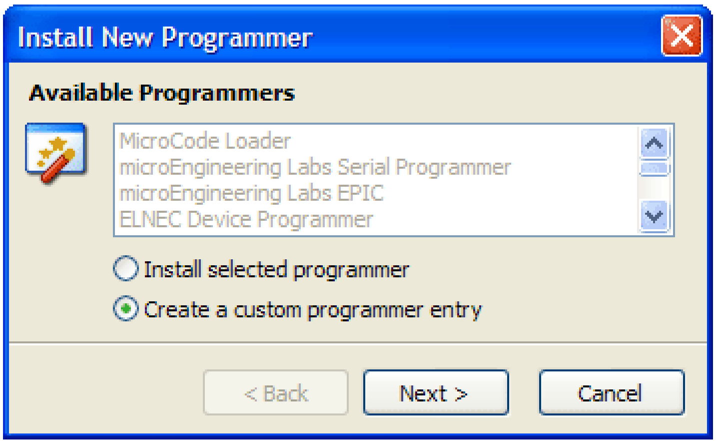
Select the programmer you want the IDE to use, then choose the Next button. The IDE will now search your computer until it locates the required executable. If your programmer is not in the list, you will need to create a custom programmer entry.
Your programmer is now ready for use. When you press the Compile and Program button on the main toolbar, you program is compiled and the programmer software started. The *.hex filename 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



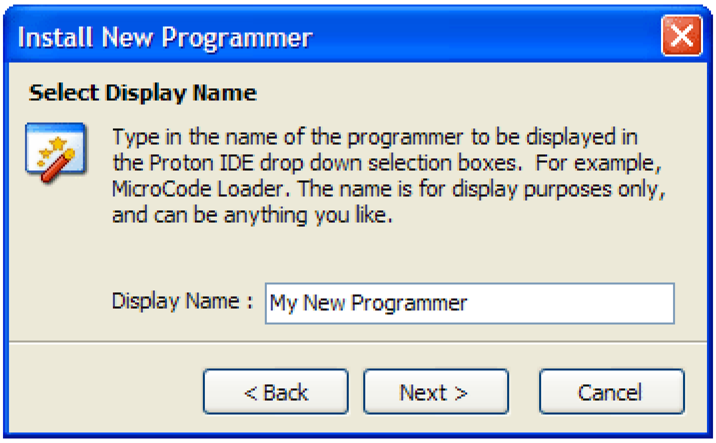
Creating a custom Programmer Entry

In most cases, the 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



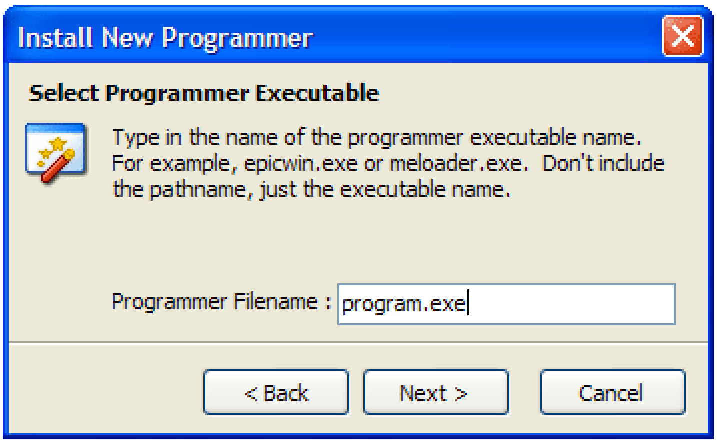
Select Display Name

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. The 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. The IDE will remember the settings for you. In the example below, the display name will be 'My New Programmer'.



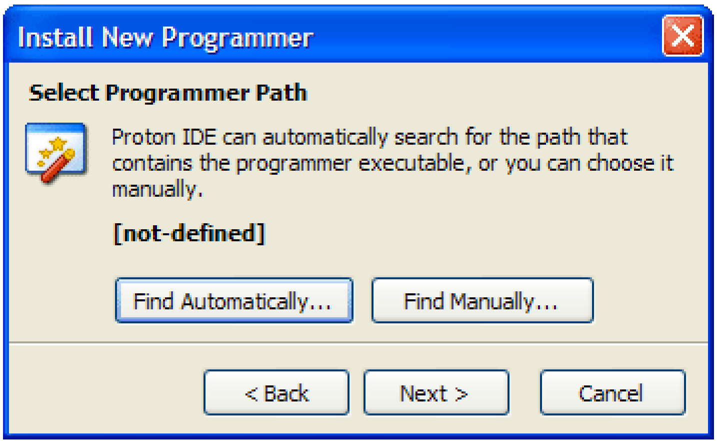
Select Programmer Executable

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.



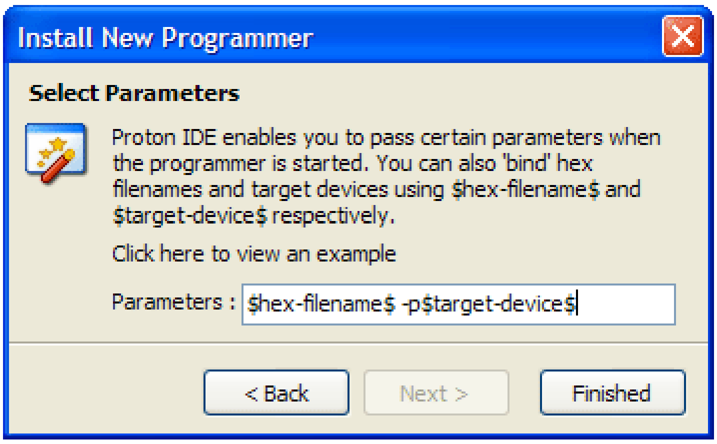
Select Programmer Path

The next screen is the path to the programmer executable. You can let the 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. The 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 IDE using a 24FJ64GA002, 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: -

-p24FJ64GA002 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

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

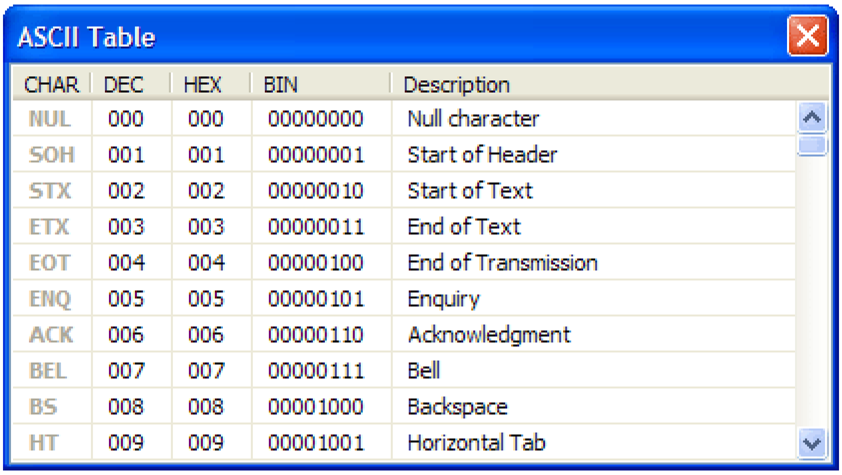
To access a plugin, select the plugin icon just above the main editor window. A drop down list of available plugins will then be displayed. Plugins can also be selected from the main menu, or by right clicking on the main editor window.

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 '$'.

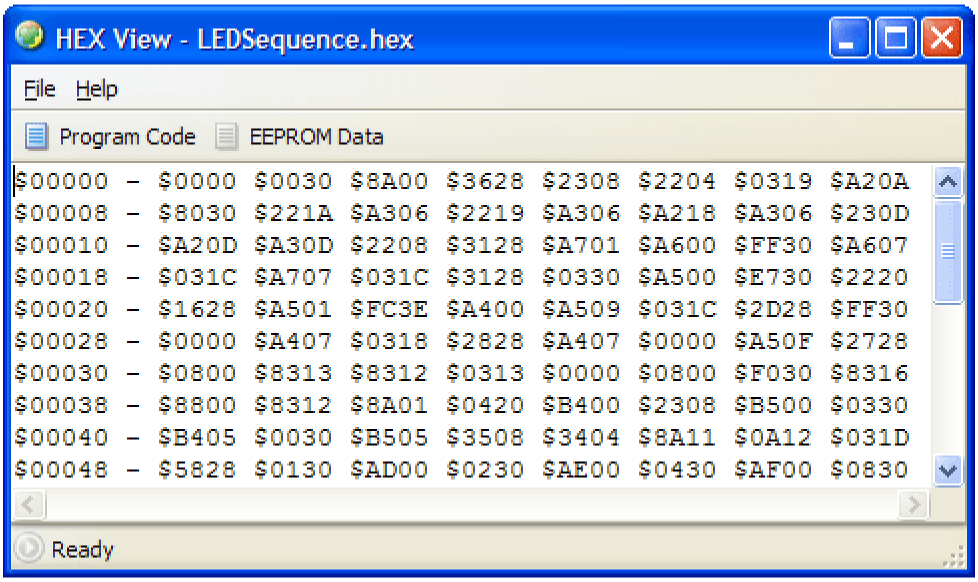


CHAR	DEC	HEX	BIN	Description
NUL	000	000	00000000	Null character
SOH	001	001	00000001	Start of Header
STX	002	002	00000010	Start of Text
ETX	003	003	00000011	End of Text
EOT	004	004	00000100	End of Transmission
ENQ	005	005	00000101	Enquiry
ACK	006	006	00000110	Acknowledgment
BEL	007	007	00000111	Bell
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 clip-board.

- **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.

Redo
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.

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, Baud rate, 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.

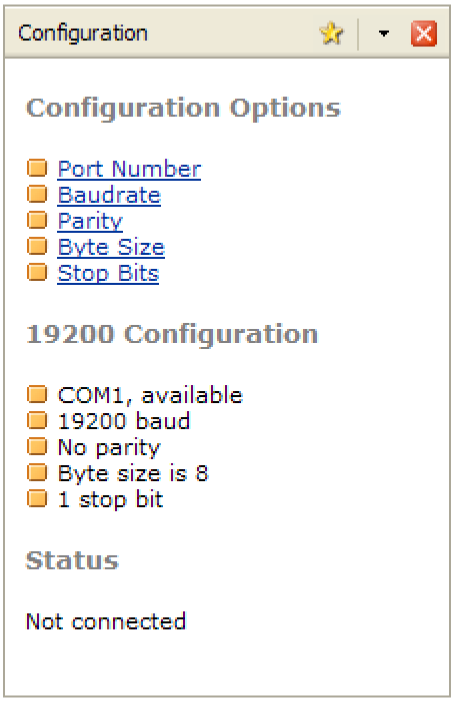
Copy
Copies any selected text from either the transmit or receive window.

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.

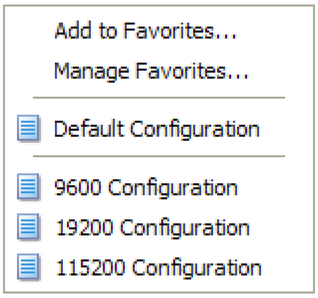
Disconnect
Disconnect the Serial Communicator from a serial port.

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.

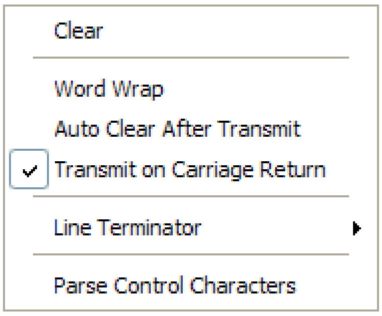


Add to Favourites
Select this option if you wish to save your current configuration. You can give your configuration a unique name, which will be displayed in the favourite drop down menu. For example, 9600 Configuration or 115200 Configuration

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.

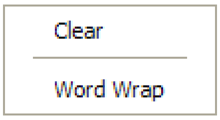
Line Terminator
You can append your data with a number of line terminations characters. These include CR, CR and LF, LF and CR, null and No Terminator.

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.

Compiler Overview

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.

Most PIC24® and dsPIC33® devices have analogue comparators and ADCs etc. When these devices first power up, the pin is set to analogue mode, which makes the pin functions work in a strange manner. To change the pins to digital, the appropriate SFRs (Special Function Registers) must be manipulated near the front of your BASIC program, or before any of the pins are accessed. The SFRs in question do have a commonality between devices, however, there are sometimes subtle differences, therefore, always read the datasheet for the device being used.

The compiler attempts to make all pins digital by manipulating the required SFRs before the user's program starts. This is accomplished within each device's ".def" file. However, this is not foolproof and some peripherals may slip through. Users are requested to inform the forum if any extra SFRs are required for a particular device, and these will be added in a later update of the compiler.

All of the microcontroller's pins are set to inputs on power-up. Once again, always read the datasheets to become familiar with the particular device being used.

Devices containing PPS (Peripheral Pin Sharing) have the ability to choose the pins used by certain peripherals. There are dedicated SFRs for PPS that must be manipulated correctly otherwise the peripheral in question will not work. See the ports section of this manual

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. Identifiers have a maximum length of 100 characters. Any identifier that breaks the 100 character limit will cause a syntax error message.

Line Labels

In order to mark statements that the program may wish to reference with the **GoTo**, **Call**, or **Gosub** 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
```

Ports and other SFRs

All of the microcontroller's SFRs (Special Function Registers), including the ports, can be accessed just like any other variable. This means that they can be read from, written to, or used in expressions directly.

```
PORTB = %0101010101010101    ' Write value to PORTB

Var1 = MyWord * PORTB ' Multiply variable MyWord with contents of PORTB
```

Remember, unlike the 8-bit microcontroller's, PIC24® and dsPIC33® devices have 16-bit wide ports and SFRs.

One thing can affect the operation of a port when used for a peripheral is PPS (Peripheral Pin Select). Most PIC24® and dsPIC33® devices have PPS which means that there is not a dedicated pin for a specific peripheral. i.e. the USART, and its pins must be designated before the peripheral can be used. The subject of PPS is too detailed for this manual, however, Microchip™ have several reference manuals that cover the use of PPS. "PIC24F® Family Reference Manual - Sect 12 - I/O" being one of them and has the file title "39711b.pdf". It is very important to read the device's datasheet and understand the operation of PPS in order for a peripheral to work correctly.

Within this manual there are several examples that use the line of code:

```
RPOR7 = 3                       ' Make PPS Pin RP14 U1TX
```

The above code designates pin RB14 (PORTB.14) for use as the TX pin for USART1.

The compiler has helper macros built in for Peripheral Pin Select, these are **PPS_Input**, **PPS_Output**, **PPS_Unlock**, and **PPS_Lock**. The macros and the related defines can be found within each device's ***.def*** file, located within the compiler's Def directory. Below, is a brief explanation of the macros, but this is not a substitute for reading the device's datasheet.

The PPS (Peripheral Pin Select) feature provides a method of enabling the user's peripheral set selection and their placement on a wide range of I/O pins. By increasing the pinout options available on a particular device, users can better tailor the microcontroller to their application, rather than trimming the application to fit the device.

The PPS feature operates over a fixed subset of digital I/O pins. Users may independently map the input and/or output of any one of many digital peripherals to any one of these I/O pins. PPS is performed in software and generally does not require the device to be reprogrammed. Optional hardware safeguards are included that prevent accidental or spurious changes to the peripheral mapping once it has been established.

Example.

```
PPS_Output(cOut_Pin_RP35, cOut_Fn_U1TX)   ' Map UART1 TX pin to RP35
PPS_Input(cIn_Pin_RPI34, cIn_Fn_U1RX)     ' Map UART1 RX pin to RPI34
```

PPS_Input (pPin, pFunction)

The **PPS_Input** macro assigns a given pin as input by configuring register RPINRx.

Not all devices use the same values for assigning pins to PPS, therefore, the parameters to use within the macros can be found in each device's .def file.

PPS_Output(pPin, pFunction)
The **PPS_Output** macro assigns a given pin as output by configuring register RPORx.

Not all devices use the same values for assigning pins to PPS, therefore, the parameters to use within the macros can be found in each device's .def file.

PPS_Lock()
The **PPS_Lock** macro performs the locking sequence for PPS assignment.

Note that this is only required if the *IOL1WAY_OFF* fuse setting is not preset within the device's configs. The compiler defaults to **not** requiring the PPS_Lock macro when manipulating PPS.

PPS_UnLock()
The **PPS_Unlock** macro performs the unlocking sequence for PPS assignment.

Note that this is only required if the *IOL1WAY_OFF* fuse setting is not preset within the device's configs. The compiler defaults to **not** requiring the PPS_Lock macro when manipulating PPS.

Numeric Representations
The compiler recognises several different numeric representations: -

Binary is prefixed by %. i.e. %01000101
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 8192) 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: -

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

Example: -

```
Print "\"Hello World\"\n\r"
```

Quoted strings of characters 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".

Standard Variables

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

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

Dim *Name* **as** *Size*

Name 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
Dim Pointy_Lrg_Rat as Double ' Create a 64-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 (System) variables for use when calculating expressions, or more complex command structures.

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, may require a certain amount of System RAM as internal variables.

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 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, PRTA1, PRTA1H, PRTA2, PRTA2H, PINM1, PINM1H, PINM2, PINM2H, BPF, BPFH.

However, if a compiler system variable is to be brought into the BASIC program for a specific reason, the reserved variables can be used, but must always be declared as a **Word** type.

RAM space required.

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

- **Double** Requires 8 bytes of RAM.
- **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, because **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 2's 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's 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's 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 one of the most versatile of the variable family types. However, more so than **Dword** types, this comes at a price because 32-bit 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.

- **Double** type variables may hold a value larger than **Float** types, and with some extra accuracy, 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 one of the most versatile of the variable family types. However, more so than **Dword** and **Float** types, this comes at a price because 64-bit 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.

Notes.

The final RAM usage will also encompass the microcontroller's stack size, therefore, even if the BASIC program only declares 4 **Byte** variables, the final RAM count will be 124. 120 bytes for the default stack size and 4 bytes for variable usage. If handled interrupts are used, the stack size will increase due to context saving and restoring requirements.

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

32-bit and 64-bit Floating Point Maths

The Proton24 compiler performs single precision (32-bit) IEEE754 Floating Point calculations and double precision (64-bit) IEEE754 Floating Point calculations.

Declaring a variable as **Float** or **Double** will enable 32-bit or 64-bit floating point calculations on that variable.

```
Dim MyFloat as Float
Dim MyDouble as Double
```

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
```

Note.

It is important to remember that 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 case of 32-bit Floats, and 8 bytes in the case of 64-bit Doubles), in essence, it is an approximation of a value. 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 and 64-bit floating point math is quite microcontroller intensive since the microcontroller is only a 16-bit processor. It also consumes quite large amounts of RAM, and code space for its operation, therefore always use floating point sparingly, and only when strictly necessary.

Unlike Proton for 8-bit microcontrollers, which uses a modified floating point format, Proton24 uses the IEEE754 standard.

An IEEE754 single precision float variable has three components: a sign bit telling whether the number is positive or negative, an exponent giving its order of magnitude, and a mantissa specifying the actual digits of the number. Below is the bit layout:

For 32-bit Float:

```
 seeeeeeeemmmmmmmmmmmmmmmmmmmmmmm  meaning
31                              0  bit number
```

For 64-bit Double:

```
 seeeeeeeeeeeeeeeemmmmmmmmmmmmmmmmmmmmmmm....  meaning
63                                          0  bit number
```

s = sign bit, e = exponent, m = mantissa

32-bit Floating Point Example Programs.

```
' Multiply two floating point values
  Device = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600     ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14  ' Select the pin for TX with USART1

  Dim MyFloat as Float
  Symbol FlNum = 1.234             ' Create a floating point constant value

  RPOR7 = 3                        ' Make PPS Pin RP14 U1TX
  MyFloat = FlNum * 10
  Hrsout Dec MyFloat, 13
  Stop

' Add two floating point variables
  Device = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600     ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14  ' Select the pin for TX with USART1

  Dim MyFloat as Float
  Dim MyFloat1 as Float
  Dim MyFloat2 as Float

  RPOR7 = 3                        ' Make PPS Pin RP14 U1TX
  MyFloat1 = 1.23
  MyFloat2 = 1000.1
  MyFloat = MyFloat1 + MyFloat2
  Hrsout Dec MyFloat, 13
  Stop

' A digital volt meter, using the on-board 10-bit ADC
  Device = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600     ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14  ' Select the pin for TX with USART1

  Declare Adin_Tad = cFRC          ' RC OSC chosen
  Declare Adin_Delay = 10          ' Allow 10us sample time

  Dim wRaw as Word
  Dim fVolts as Float
  Symbol Quanta = 3.3 / 1024   ' Calculate the quantising value for 10-bits

  RPOR7 = 3                       ' Make PPS Pin RP14 U1TX
  AD1CON2 = 0                     ' +Vref = AVdd, -Vref = AVss
  AD1PCFGbits_PCFG0 = 0           ' Analogue input on AN0
  While
    wRaw = Adin 0
    fVolts = wRaw * Quanta
    Hrsout Dec2 fVolts, "V\r"
    DelayMs 300
  Wend
```

64-bit Floating Point Example Programs.

```
' Multiply two 64-bit floating point values
  Device = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600      ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14   ' Select the pin for TX with USART1

  Dim MyDouble as Double
  Symbol FlNum = 1.234              ' Create a floating point constant value

  RPOR7 = 3                         ' Make PPS Pin RP14 U1TX
  MyDouble = FlNum * 10
  Hrsout Dec MyDouble, 13
  Stop

' Add two 64-bit floating point variables
  Device = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600      ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14   ' Select the pin for TX with USART1

  Dim MyDouble as Double
  Dim MyDouble 1 as Double
  Dim MyDouble 2 as Double

  RPOR7 = 3                         ' Make PPS Pin RP14 U1TX
  MyDouble1 = 1.23
  MyDouble2 = 1000.1
  MyDouble = MyDouble1 + MyDouble2
  Hrsout Dec MyDouble, 13
  Stop

' A digital volt meter, using the on-board 10-bit ADC
  Device = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600       ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14    ' Select the pin for TX with USART1

  Declare Adin_Tad = cFRC           ' RC OSC chosen
  Declare Adin_Delay = 10           ' Allow 10us sample time

  Dim wRaw as Word
  Dim fVolts as Double
  Symbol Quanta = 3.3 / 1024   ' Calculate the quantising value for 10-bits

  RPOR7 = 3                         ' Make PPS Pin RP14 U1TX
  AD1CON2 = 0                       ' +Vref = AVdd, -Vref = AVss
  AD1PCFGbits_PCFG0 = 0             ' Analogue input on AN0
  While
    wRaw = Adin 0
    fVolts = wRaw * Quanta
    Hrsout Dec2 fVolts, "V\r"
    DelayMs 300
  Wend
```

Notes.
Any expression that contains a floating point variable or constant will always be calculated as a floating point, even if the expression also contains integer constants or variables. The same applies for Doubles. If an expression has a mix of 32-bit floats and 64-bit doubles, the it will be carried out as 64-bit Double.

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 = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600      ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14   ' Select the pin for TX with USART1

  Dim MyDword as Dword
  Dim MyFloat as Float
  Symbol cPI = 3.14

  RPOR7 = 3                         ' Make PPS Pin RP14 U1TX
  MyFloat = 10
'
' Float calculation with result 13.14, reduced to integer 13
'
  MyDword = MyFloat + cPI
  Hrsout Dec MyDword, 13            ' Display the integer result 13
```

Floating Point To Integer Rounding
Assigning a 32-bit or 64-bit floating point variable to an integer type will be truncated to the lowest value by default. For example:

```
  MyFloat = 3.9
  MyDword = MyFloat
```

The variable MyDword will hold the value of 3.

Floating Point Exception Flags

The floating point exception flags are accessible from within the BASIC program via the system variable _FP_FLAGS.

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.

See also : Dim, Symbol, Aliases, Arrays.

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**, **HighSByte**, **LowSByte**, **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 MyWord as Word           ' Declare an unsigned Word variable
  Dim MyWord_Hi as MyWord.HighByte
' MyWord_Hi now represents the unsigned high byte of variable MyWord
```

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

HighSByte 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 MyWord as SWord          ' Declare a signed Word variable
  Dim MyWord_Hi as MyWord.SByte1
' MyWord_Hi now represents the signed high byte of variable MyWord
```

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

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. **HighSByte** will still extract the signed high byte of the variable, as will **SByte3**.

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

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

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

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
```

Finer points of variable handling.

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 MyWord as Word
```

Will produce the assembler code: -

```
  MyWord: .space 2
.def MyWord
  .val MyWord
  .scl 2
  .size 2
  .type 016
.endef
.set MyWordH, n
```

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

```
  Mov.b #1,W0
  Mov.b WREG, MyWordH      ' Load the high byte of MyWord 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 MyDword as Dword
```

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

```
  MyDword.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 sMyDword as SDword   ' Declare a 32-bit signed variable

  sMyDword.Byte3 = 1       ' Load the unsigned high byte with the value 1
  sMyDword.SByte0 = -1     ' Load the signed low byte with the value -1
  sMyDword.SWord0 = 128    ' Load signed low and mid1 bytes with 128
```

Notes.

The final RAM usage will also encompass the microcontroller's stack size, therefore, even if the BASIC program only declares 4 **Byte** variables, the final RAM count will be 84. 80 bytes for the default stack size and 4 bytes for variable usage. If handled interrupts are used, the stack size will increase due to context saving and restoring requirements.

RAM locations for variables is allocated automatically within the microcontroller because the PIC24® and dsPIC33® range of devices have specific requirements concerning RAM addressing. Which are:

- 16-bit variables must be located on a 16-bit RAM address boundary.
- 32-bit and 64-bit variables must be placed on a 16-bit RAM address boundary, but should be placed on a 32-bit RAM address, if possible, for more efficiency with some mnemonics.
- 8-bit variables can be located on an 8-bit,16-bit or 32-bit RAM address boundary.

Therefore, the order of variable placements is:

- The microcontroller's 16-bit stack is located before all variables are placed.
- The compiler's 16-bit system variables are placed.
- **Word** variables are placed.
- **Dword** variables are placed.
- **Float** variables are placed.
- **Double** variables are placed.
- **Byte** variables are placed.
- **Word Arrays** are placed.
- **Dword Arrays** are placed.
- **Float Arrays** are placed.
- **Byte Arrays** are placed.
- **String** variables are placed.

The logic behind the variable placements is because of the microcontroller’s near and far RAM.

The first 8192 bytes of RAM are considered "near" RAM, while space above that is considered "far" RAM. By default, the compiler sets all user variables to near RAM. However, when near RAM space is full, the compiler will place variables in far RAM (above 8192).

The special significance of near versus far to the compiler is that near RAM accesses are encoded in only one mnemonic using direct addressing, while accesses to variables in far RAM require two to three mnemonics using indirect addressing.

Standard variables are used more commonly within a BASIC program, therefore should reside in near RAM for efficiency. Arrays and Strings are generally accessed indirectly anyway, therefore, it is usually of little consequence if they reside in near or far RAM.

The PIC24® and dsPIC33® range of devices have 16 WREG SFRs (Special Function Registers), each 16-bits wide. These are invaluable assets when the fastest possible speed is required by a program's routines or procedures. However, the compiler needs certain WREG SFRs for its own use, as does the microcontroller itself. Below is a rough list of the compiler's and microcontroller's WREG use.

- WREG0, WREG1, WREG2 and WREG3 are used internally by the compiler for its mathematical expressions, as well as comparisons. These WREGs should be considered as volatile and never used for storage of data for more than a brief length of time.

- WREG6, WREG7 and WREG8 are sometimes used as parameter storage for the compiler's commands.

- WREG12 and WREG13 are used for array indexing.

- WREG14 is the microcontroller's Frame Pointer. As such, it must not be accessed directly unless the user is fully aware of the ramifications.

- WREG15 is the microcontroller's Stack Pointer. As such, it must not be accessed directly unless the user is fully aware of the ramifications.

A WREG SFR can be used in a BASIC program just the same as any user defined variable. For example:

```
WREG0 = 12345
```

Note that the WREG SFRs are each 16-bits wide, but they can be sliced by the **.Byte0** and **.Byte1** directives, just as a user variable can be:

```
WREG0.Byte0 = 12
WREG0.Byte1 = 34
```

Symbols

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

```
Symbol cCat = 123
Symbol cTiger = cCat                ' cTiger now holds the value of cCat
Symbol cMouse = 1
Symbol cTigOuse = cTiger + cMouse ' Add cTiger to cMouse to make cTigOuse
```

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 = 3.3 / 1024
```

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

```
Symbol MyCon = (PORTB + 1)' MyCon will hold the value 715 (714+1)
```

Symbol is also useful for aliasing Ports and SFRs (Special Function Registers): -

```
Symbol LED = PORTA.1       ' LED now references bit-1 of PORTA
Symbol OSCFAIL = INTCON1.1' OSCFAIL now refers to bit-1 of INTCON1 SFR
```

Creating and using Arrays

The Proton24 compiler supports multi part **Byte**, **Word**, **Dword**, **SByte**, **SWord**, **SDword** and **Float** variable arrays. 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
Dim Name[length] as Float
```

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.
Dim fMyArray[10] as Float ' Create a 10 element floating point array.
```

Arrays may have up to 65535 elements.

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
Hrsout "MyArray[3] = ", Dec MyArray[3], 13
```

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

```
Device = 24FJ64GA002
Declare Xtal = 16
Declare Hserial_Baud = 9600     ' USART1 baud rate
Declare Hrsout1_Pin = PORTB.14  ' Select the pin for TX with USART1

Dim MyArray[10] as Byte          ' Create a 10-byte array.
Dim Index as Byte                ' Create a Byte variable.

RPOR7 = 3                        ' Make PPS Pin RP14 U1TX
For Index = 0 to 9               ' Repeat with Index= 0,1,2...9
  MyArray[Index] = Index * 10 ' Write to each element of the array.
Next
For Index = 0 to 9               ' Repeat with Index= 0,1,2...9
  Hrsout Dec MyArray[Index], 13 ' Show the contents of each element.
  DelayMs 500                    ' Wait long enough to view the values
Next
```

If the previous 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 interpreted 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 previous example, 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 prevent this from happening.

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

```
Device = 24FJ64GA002
Declare Xtal = 16
Declare Hserial_Baud = 9600      ' USART1 baud rate
Declare Hrsout1_Pin = PORTB.14  ' Select the pin for TX with USART1

Dim MyArray[10] as Byte          ' Create a 10-byte array.
Dim Index as Byte                ' Create a Byte variable.
RPOR7 = 3                        ' Make PPS Pin RP14 U1TX
For Index = 0 to 8               ' Repeat with Index= 0,1,2...8
  MyArray[Index + 1] = Index * 10 ' Write to each element of array
Next
For Index = 0 to 8               ' Repeat with Index= 0,1,2...8
  Hrsout Dec MyArray[Index + 1], 13 ' Show the contents of elements
  DelayMs 500                    ' Wait long enough to view the values
Next
```

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 MyByte 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
MyByte = 10                ' Variable MyByte now holds the value 10
MyArray[Index] = 20        ' Load the 6th element of MyArray with value 20
Result = (MyByte * MyArray[Index]) / 20' Do a simple expression
Hrsout Dec Result, 13      ' Display result of expression
```

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

(10 * 20) / 20

MyByte 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.

Byte Arrays as Strings

Byte 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.

Some of 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, MsbPre, [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.

Finer points of array variables.

When an array is created, the compiler creates each of its elements as an independent variable that has the same width and sign as the parent array. For example:

```
Dim MyByteArray[10] as Byte
```

Will also produce 10 separate variables named:

MyByteArray_0
MyByteArray_1
MyByteArray_2
MyByteArray_3
MyByteArray_4
MyByteArray_5
MyByteArray_6
MyByteArray_7
MyByteArray_8
MyByteArray_9

Notice the underscore after the name of the array, and preceding the element's positional value within the array. Each of these elements is an unsigned **Byte** variable in it's own right, and can be accessed collectively or independently. The same principle applies to all array types supported by the compiler.

Note that this differs from Proton for 8-bit devices in that the separation between the array name and the element number is a hash '# 'in Proton 8-bit, but this is not allowed in the 16-bit assembler, therefore, they are underscores in Proton24.

Block Array Assigning

One array can be loaded into another array by issuing the names of the arrays but without the square brackets. For example:

```
Dim SourceArray[10] as Byte = 1,2,3,4,5,6,7,8,9,10
Dim DestArray[10] as Byte

DestArray = SourceArray ' Copy the contents of SourceArray into DestArray
```

If different type arrays are used as the assignment or the source, truncation or extrapolation will take place. If the assignment array has fewer elements than the source array, only the elements that will fit into the assignment array will be copied.

Creating and using String variables

A string variable is essentially a byte array that is terminated by a null (represented by 0). A string is intended to hold only ASCII characters.

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 8192, allowing up to 8192 characters to be stored.

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

```
Dim MyString as String * 20
```

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

```
  Device = 24FJ64GA002
  Declare Xtal = 16
'
' 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".

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 = 24FJ64GA002
  Declare Xtal = 16

  Dim DestString as String * 20    ' Create a String for 20 characters
  Dim SourceString as String * 20  ' Create another String for 20 characters

  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"
```

Note that Strings cannot be subtracted, multiplied or divided, and cannot be used as part of a standard 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 24FJ64GA002
  Declare Xtal = 16

  Dim DestString as String * 20    ' Create a String for 20 characters
  Dim SourceString as String * 20  ' Create another String
  Dim CodeStr as Code = "World",0

  SourceString = "Hello "          ' Load the string with characters
  DestString = SourceString + Cstr CodeStr  ' Concatenate the string
  Print DestString               ' Display the result which is "Hello World"
```

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 24FJ64GA002
  Declare Xtal = 16

  Dim DestString as String * 20        ' Create a String for 20 characters
  Dim SourceString as String * 20      ' Create another String
  Dim CodeStr as Code = "World",0

  SourceString = "Hello "            ' Load the string with characters
  DestString = SourceString + CodeStr  ' Concatenate the string
  Print DestString                   ' Display the result which is "Hello World"
```


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 24F08KL200                  ' Choose a device with on-board eeprom
  Declare Xtal = 16

  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"
```


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 24FJ64GA002
  Declare Xtal = 16
  Dim DestString as String * 30    ' Create a String
  Dim SourceString as String * 20  ' Create another String
  Dim MyWord as Word               ' Create a Word variable

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

Left$ Example

```
' Copy 5 characters from the left of SourceString
' and add to a quoted character string
  Device 24FJ64GA002
  Declare Xtal = 16
  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"
```

Right$ Example

```
' Copy 5 characters from the right of SourceString
' and add to a quoted character string
  Device 24FJ64GA002
  Declare Xtal = 16
  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"
```

Mid$ Example

```
' Copy 5 characters from position 4 of SourceString
' and add to quoted character strings
  Device 24FJ64GA002
  Declare Xtal = 16
  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"
```

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 24FJ64GA002
  Declare Xtal = 16
  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"
```

ToLower Example

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

  Device 24FJ64GA002
  Declare Xtal = 16
  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"
```

Loading a String Indirectly

If the Source String is a signed 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 24FJ64GA002
  Declare Xtal = 16
  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"
```


Slicing a String.

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

```
Device 24FJ64GA002
Declare Xtal = 16
Dim SourceString as String * 20  ' Create a String

SourceString[0] = "H"   ' Place letter "H" as first character
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"
```

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 24FJ64GA002
  Declare Xtal = 16
  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"
```

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 24FJ64GA002
  Declare Xtal = 16
  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)
```


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 code memory strings,**
Creating and using code memory strings,
Len, Left$, Mid$, Right$
String Comparisons, Str$, ToLower, ToUpper, AddressOf.

Procedures

A procedure is essentially a subroutine in a wrapper that can be optionally passed parameter variables and optionally return a variable. The code within the procedure block is self contained, including local variables, symbols and labels who's names are local to the procedure's block and cannot be accessed by the main program or another procedure, even though the names may be the same within different procedures.

The Proton24 compiler has a rudimentary procedure mechanism that allows procedures to be constructed along with their local variables and, moreover, the procedure will not be included into the program unless it is called by the main program or from within another procedure. A procedure also has the ability to return a variable for use within an expression or comparison etc. This means that libraries of procedures can be created and only the ones called will actually be used.

A procedure is created by the keyword **Proc** and ended by the keyword **EndProc**

A simple procedure block is shown below:

```
Proc MyProc(pBytein as Byte)
  Hrsout Dec pBytein, 13
EndProc
```

To use the above procedure, give its name and any associated parameters:

```
MyProc(123)
```

Parameters

A procedure may have up to 10 parameters. Each parameter must be given a unique name and a variable type. The parameter name must consist of more than one character. The types supported as parameters are:

Bit, **Byte**, **SByte**, **Word**, **SWord**, **Dword**, **SDword**, **Float**, **Double**, and **String**.

A parameter can be passed by value or by reference. By value will copy the contents into the parameter variable, while by reference will copy the address of the original variable into the parameter variable. By default, a parameter is passed by value. In order to pass a parameter by reference, so that it can be accessed by one of the **PtrX** commands, the parameter name must be preceded by the text **ByRef**. For clarification, the text **ByValue** may precede a parameter name to illustrate that the variable is passed by value. For example:

```
Proc MyProc(ByRef pWordin as Word, ByValue pDwordin as Dword)
  Hrsout  Dec pWordin, " : ", Dec pDwordin, 13
EndProc
```

The syntax for creating parameters is the same as when they are created using **Dim**. String or Array variables must be given lengths. For example, to create a 10 character **String** parameter use:

```
Proc MyProc(pMyString as String * 10)
```

To create a 10 element unsigned **Word** array parameter, use:

```
Proc MyProc(pMyArray[10] as Word)
```

A parameter can also be aliased to an SFR (Special Function Register). For example:

```
Proc MyProc(pMyWord as WREG0)
```

As with standard variables, the aliased parameter can also be casted to a type that has fewer bytes. For example, a **Word** variable can be casted to a **Byte** type, or a **Dword** can be casted to a **Word** or **Byte** type etc…

```
Proc MyProc(pMyByte as WREG2.Byte0)
```

Local Variable and Label Names

Any label, constant or variable created within a procedure is local to that procedure only. Meaning that it is only visible within the procedure, even if the name is the same as other variables created in other procedures, or global constants or variables. A local variable is created exactly the same as global variables. i.e. using **Dim**:

```
Proc MyProc(pMyByte as Byte)
Dim MyLocal as Byte          ' Create a local byte variable
  MyLocal = pMyByte          ' Load the local variable with parameter variable
EndProc
```

Note that a local variable's name must consist of more than 1 character.

Return Variable

A procedure can return a variable of any type, making it useful for inclusion within expressions. The variable type to return is added to the end of the procedure's template. For example:

```
Proc MyProc(), SByte
  Result = 10
EndProc
```

All variable types are allowed as return parameters and follow the same syntax rules as **Dim**. Note that a return name is not required, only a type. For example:

```
Proc MyProc(), [12] as Byte    ' Procedure returns a 12 element byte array
Proc MyProc(), [12] as Word    ' Procedure returns a 12 element word array
Proc MyProc(), [12] as Dword   ' Procedure returns a 12 element dword array
Proc MyProc(), [12] as Float   ' Procedure returns a 12 element float array
Proc MyProc(), String * 12     ' Procedure returns a 12 character string
```

In order to return a value, the text **"Result"** is used. Internally, the text **Result** will be mapped to the procedure's return variable. For example:

```
Proc MyProc(pBytein as Byte), Byte
  Result = pBytein ' Transfer the parameter directly to the return variable
EndProc
```

The **Result** variable is mapped internally to a variable of the type given as the return parameter, therefore it is possible to use it the same as any other local variable, and upon return from the procedure, its value will be passed. For example:

```
Proc MyProc(pBytein as Byte), Byte
  Result = pBytein        ' Transfer the parameter to the return variable
  Result = Result + 1     ' Add one to it
EndProc
```


Returning early from a procedure is the same as returning from a subroutine. i.e. using the **Return** keyword.

```
Proc MyProc(pBytein as Byte), Byte
  Result = pBytein           ' Transfer the parameter to the return variable
  If pBytein = 0 Then Return ' Perform a test and return early if required
  Result = Result + 1        ' Otherwise… Add one to it
EndProc
```

A return parameter can also be aliased to an SFR (Special Function Register). For example:

```
Proc MyProc(), WREG0
```

As with standard variables, the aliased return parameter can also be casted to a type that has fewer bytes. For example, a **Word** variable can be casted to a **Byte** type, or a **Dword** can be casted to a **Word** or **Byte** type etc…

```
Proc MyProc(), WREG2.Byte0
```

Below is an example procedure that mimics the compiler's 16-bit **Dig** command.

```
  Device = 24EP128MC202
  Declare Xtal = 140.03
  Declare Hserial_Baud = 9600      ' UART1 baud rate
  Declare Hrsout1_Pin = PORTB.11   ' Select pin to be used for USART1 TX

  Dim MyWord As Word = 12345
'--------------------------------------------------
' Emulate the 16-bit Dig command's operation
' Input      : pWordin holds the value to extract from
'            : pDigit holds which digit to extract (1 To 5)
' Output     : Result holds the extracted value
' Notes      : None
'
Proc DoDig16(pWordin As Word, pDigit As Byte), Byte
Dim DigitLoop As Byte

  pWordin = Abs pWordin
  If pDigit > 0 Then
    For DigitLoop = (pDigit - 1) To 0 Step -1
      pWordin = pWordin / 10
    Next
  EndIf
  Result = pWordin // 10
EndProc
'--------------------------------------------------
Main:
' Setup the Oscillator to operate the device at 140.03MHz
' Fosc = (7.37 * 76) / (2 * 2) = 140.03MHz
  PLL_Setup(76, 2, 2, $0300)
  RPOR4.Byte1 = 1                          ' Make PPS Pin RB11 U1TX

  HRSOut Dec DoDig16(MyWord, 0)
  HRSOut Dec DoDig16(MyWord, 1)
  HRSOut Dec DoDig16(MyWord, 2)
  HRSOut Dec DoDig16(MyWord, 3)
  HRSOut Dec DoDig16(MyWord, 4), 13
```

```
'-------------------------------------------------------------------------
' Configure for internal 7.37MHz oscillator with PLL
' OSC pins are general purpose I/O
'
    Config FGS = GWRP_OFF, GCP_OFF
    Config FOSCSEL = FNOSC_FRCPLL, IESO_ON, PWMLOCK_OFF
    Config FOSC = POSCMD_NONE, OSCIOFNC_ON, IOL1WAY_OFF, FCKSM_CSDCMD
    Config FWDT = WDTPOST_PS256, WINDIS_OFF, PLLKEN_ON, FWDTEN_OFF
    Config FPOR = ALTI2C1_ON, ALTI2C2_OFF
    Config FICD = ICS_PGD1, JTAGEN_OFF
```

Notes

The compiler's implementation of procedures is not as thorough as a true procedural language such as C or Pascal because they have had to be added to an already flat language. However, they are still a powerful feature of the language when used appropriately. Procedures are not supported in every instance of the compiler and if one is not supported within a particular command, a syntax error will be produced. In which case, an intermediate variable will need to be created to hold the procedure's return result:

```
MyTemp = MyProc()
```

The compiler does not re-cycle RAM for parameters or local variables.

A parameter that is passed **ByRef** can only ever be a **Byte**, **Word** or **Dword** type, because it will hold the address of the variable passed to it and not its value. This is then used by either **Ptr8**, **Ptr16**, **Ptr32** and **Ptr64** in order to manipulate the address indirectly. An example of this mechanism is shown below:

```
' Demonstrate a procedure for finding the length of a word array
' given a particular terminator value
'
  Device = 24FJ64GA002
  Declare Xtal = 16
'
' USART1 declares
'
  Declare Hserial_Baud = 9600       ' UART1 baud rate
  Declare Hrsout1_Pin = PORTB.14    ' Select the pin for TX with USART1

  Dim MyLength As Word
  Dim MyArray[20] As Word = 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,0

'--------------------------------------------------------------------------
' Find the length of a word array with a user defined terminator
' Input     : pArrayIn holds the address of the word array
'           : pTerminator holds the terminator value
' Output    : Returns the length of the word array upto the terminator
' Notes     : Uses indirect addressing using ByRef and Ptr16
'
Proc LengthOf(ByRef pInAddress As Word, pTerminator As Word), Word
  Result = 0                        ' Clear the result of the procedure
  While                             ' Create an infinite loop
    '
    ' Increment up the array and exit the loop when the terminator is found
    '
    If Ptr16(pInAddress++) = pTerminator Then Break
```

```
    Inc Result                        ' Increment the count
  Wend
EndProc
'-----------------------------------------------------------------------------
Main:
  RPOR7 = 3                             ' Make PPS Pin RP14 U1TX
'
' Find the length of a null terminated word array
'
  MyLength = LengthOf(MyArray, 0)
  HRSOut Dec MyLength, 13   ' Display the result on a serial terminal
```

See Also: Ptr8, Ptr16, Ptr32, Ptr64

A Typical Flat 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 or declare being created after it is supposed to be used.

The recommended layout for a program is shown below.

```
  Device                          ' Always required
'
  Xtal declare                    ' Always required
'
  General declares
'
  Includes
'
  Constants and/or Variables
'
  GoTo Main                         ' Jump over the subroutines (if any)
'
  Subroutines go here
'
Main:
  Main Program code goes here
```

For example:

```
  Device = 24FJ64GA002
'-----------------------------------------
  Declare Xtal = 32
  Declare Hserial_Baud = 9600
'-----------------------------------------
' Load an include file (if required)
  Include "MyInclude.inc"
'-----------------------------------------
' Create Variables
  Dim MyWord as Word              ' Create a Word size variable
'-----------------------------------------
' Define Constants and/or aliases
  Symbol MyConst = 10             ' Create a constant
'-----------------------------------------
  GoTo Main                       ' Jump over the subroutine/s (if any)
'-----------------------------------------
' Simple Subroutine
AddIt:
  MyWord = MyWord + MyConst       ' Add the constant to the variable
  Return                          ' Return from the subroutine
'-----------------------------------------
' Main Program Code
Main:
  RPOR7 = 3                       ' Make PPS Pin RP14 U1TX
  MyWord = 10                     ' Pre-load the variable
  GoSub AddIt                     ' Call the subroutine
  Hrsout Dec MyWord, 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.

A Typical Procedural 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 or declare being created after it is supposed to be used.

The recommended layout for a program is shown below.

```
  Device                        ' Always required
'
  Xtal declare                  ' Always required
'
  General declares
'
  Includes
'
  Constants and/or Variables
'
  Procedures go here
'
Main:
  Main Program code goes here
```

For example:

```
  Device = 24FJ64GA002
'-----------------------------------------
  Declare Xtal = 32
  Declare Hserial_Baud = 9600
'-----------------------------------------
' Load an include file (if required)
  Include "MyInclude.inc"
'-----------------------------------------
' Create Variables
  Dim MyWord as Word            ' Create a Word size variable
'-----------------------------------------
' Define Constants and/or aliases
  Symbol MyConst = 10           ' Create a constant
'-----------------------------------------
' Simple Procedure
Proc AddIt(pMyWord1 as Word, pMyWord2 as Word), Word
  Result = pMyWord1 + pMyWord2 ' Add the two variables as the result
EndProc
'-----------------------------------------
' Main Program Code
Main:
  RPOR7 = 3                     ' Make PPS Pin RP14 U1TX
  MyWord = 10                   ' Pre-load the variable
  MyWord = AddIt(MyWord, MyConst)  ' Call the procedure
  Hrsout Dec MyWord, 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.

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 '**:**'.

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

Multiple-line version: -

```
Low PORTB                  ' Make all pins on PORTB outputs
For MyByte = 0 to 100      ' Count from 0 to 100
  PORTB = MyByte           ' Make PORTB = MyByte
Next                       ' Continue counting until 100 is reached
```

Single-line version: -

```
Low PORTB: For MyByte = 0 to 100 : PORTB = MyByte: Next
```

Line Continuation Character '_'

Lines that are too long to display. i.e. greater than 1000 characters, may be split using the continuation character '_'. This will direct the continuation of a command to the next line. Its 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 Code Memory Tables

All 24-bit core devices have the ability to read their own flash 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 form of data storage and retrieval, the **Dim as Code** directive proves this, as it uses the mechanism of reading and storing in the microcontroller's flash memory.

```
Dim MyCode as Code = As Dword 1, 2, 3, 4, 5
```

or

```
Dim MyCode as PSV = As Word 100, 200, 300, 400
```

Both of the above lines of code will create a data table in the device's code memory, however, the **PSV** directive will ensure that the **AddressOf** function returns the PSV address of the table, instead of its actual code memory address. This is used mainly for DSP operations.

The data produced by the **Code** or **PSV** directives follows the same casting rules as the Cdata directive, in that the table's data can be given a size that each element will occupy.

```
Dim CodeString As Code = "Hello World", 0
```

The above line will create, in code memory, the values that make up the ASCII text "Hello World", at address CodeString. Note the null terminator after the ASCII text.

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

```
  Hrsout CodeString
```

The label that declared the address where the list of code memory values resided now becomes the string's name.

Note the null terminators after the ASCII text in the table data. Without these, the microcontroller will continue to transmit data until it reaches a 0 value within code.

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

Using the **Cstr** modifier it is also possible to use constants, variables and expressions that hold the address of the code memory data (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 in code memory.

Example

```
  Device 24FJ64GA002
  Declare Xtal = 16

  Dim Address as Word                 ' Address holding variable
'
' Create the text to display
'
  Dim CodeString1 as Code = "Hello ", 0
  Dim CodeString2 as Code = "World", 0

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

  Address = AddressOf(CodeString1) ' Point address to CodeString1
  Print Cstr Address                  ' Display CodeString1
  Address = AddressOf(CodeString2) ' Point Address to CodeString2
  Print Cstr Address                  ' Display CodeString2
```


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.

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
  Hrsout "Equal\r"
Else
  Hrsout "Not Equal\r"
EndIf
```

But as you've found out if you read the *Creating Strings* section, there is more than one type of **String** in a PIC24® and dsPIC33® microcontroller. There is a RAM **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 = 24FJ64GA002
  Declare Xtal = 16

  Dim String1 as String * 20   ' Create a String
  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
```

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 = 24FJ64GA002
  Declare Xtal = 16

  Dim String1 as String * 20   ' Create a String

  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
```

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 = 24FJ64GA002
  Declare Xtal = 16

  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

  RPOR7 = 3                    ' Make PPS Pin RP14 U1TX
  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"
  Hrsout DestString, 13        ' Display DestString
```

Example 4

```
' Compare a string variable to a string held in code memory
  Device = 24FJ64GA002
  Declare Xtal = 16

  Dim String1 as String * 20      ' Create a String
  Dim CodeString as Code = "EGGS", 0

  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
```

Example 5

```
' String comparisons using Select-Case
  Device = 24FJ64GA002
  Declare Xtal = 16

  Dim String1 as String * 20      ' Create a String for 20 characters

  RPOR7 = 3                       ' Make PPS Pin RP14 U1TX
  String1 = "EGGS"                ' Pre-load String String1 with the text EGGS
  Select String1                  ' Start comparing the string
    Case "EGGS"                   ' Is String1 equal to EGGS?
      Hrsout "Found EGGS\r"
    Case "BACON"                  ' Is String1 equal to BACON?
      Hrsout "Found BACON\r"
    Case "COFFEE"                 ' Is String1 equal to COFFEE?
      Hrsout "Found COFFEE\r"
    Case Else                     ' Default to...
      Hrsout "No Match\r"         ' Displaying no match
    EndSelect
```

See also : **Creating and using Strings**
Creating and using code memory strings
If-Then-Else-EndIf, Repeat-Until
Select-Case, While-Wend .

Relational Operands

Relational operands 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 operands accepted by the compiler:

Operator	Relation	Expression Type
=	Equality	X = Y
==	Equality	X == Y (Same as above Equality)
<>	Inequality	X <> Y
<	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 Operands

The operands **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**: -

```
  Device = 24FJ64GA002
  Declare Xtal = 16
  Dim MyByte1 as Byte
  Dim MyByte2 as Byte
  RPOR7 = 3                   ' Make PPS Pin RP14 U1TX

  MyByte1 = 5
  MyByte2 = 9
  If MyByte1 = 5 And MyByte2 = 10 Then GoTo Res_True
  Stop
Res_True:
  Hrsout "Result is True.\r"
```

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.

Parenthesis (or rather the lack of it!).

Every compiler has it's quirky rules, and the Proton24 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 Proton24 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 Proton24 compiler performs all math operations in full hierarchal order. Which means that there is precedence to the operands. 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 operands 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.
Remainder '//'.	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.
Abs.	Returns the absolute value of a signed value.
Acos	Returns the Arc Cosine of a 32-bit floating point value in radians.
Asin	Returns the Arc Sine of a 32-bit floating point value in radians.
Atan	Returns the Arc Tangent of a 32-bit floating point value in radians.
Ceil	Returns the ceiling of a 32-bit floating point value.
Cos.	Returns the Cosine of a 32-bit floating point value in radians.
Dcd.	2 n -power decoder of a four-bit integer value.
Dig '?'.	Returns the specified decimal digit of a positive integer value.
Exp	Deduce the exponential function of a 32-bit floating point value.
Floor	Returns the floor of a 32-bit floating point value.
fAbs	Returns the absolute value of a 32-bit or 64-bit floating point value.
ISin	Returns the integer Sine of an integer value in radians.
ICos	Returns the integer Cosine of an integer value in radians.
ISqr	Returns the integer Square Root of an integer value.
Log	Returns the Natural Log of a 32-bit floating point value.
Log10	Returns the Log of a 32-bit floating point value.
Modf	Split a 32-bit floating point value into its fractional and whole parts.
Ncd.	Priority encoder of a 16-bit integer value.
Pow	Computes a 32-bit floating point variable to the power of another.
Rev '@'.	Reverses the order of the lowest bits in an integer value.
Sin.	Returns the Sine of a 32-bit floating point value in radians.
Sqr.	Returns the Square Root of a 32-bit floating point value.
Tan	Returns the Tangent of a 32-bit floating point value in radians.

dAcos	Returns the Arc Cosine of a 64-bit floating point value in radians.
dAsin	Returns the Arc Sine of a 64-bit floating point value in radians.
dAtan	Returns the Arc Tangent of a 64-bit floating point value in radians.
dCeil	Returns the ceiling of a 64-bit floating point value.
dCos.	Returns the Cosine of a 64-bit floating point value in radians.
dExp	Deduce the exponential function of a 64-bit floating point value.
dFloor	Returns the floor of a 64-bit floating point value.
dLog	Returns the Natural Log of a 64-bit floating point value.
dLog10	Returns the Log of a 64-bit floating point value.
Modd	Split a 64-bit floating point value into its fractional and whole parts.
dPow	Computes a 64-bit floating point variable to the power of another.
dSin.	Returns the Sine of a 64-bit floating point value in radians.
dSqr.	Returns the Square Root of a 64-bit floating point value.
dTan	Returns the Tangent of a 64-bit floating point value in radians.

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.

Operands
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.

```
  Device = 24FJ64GA002
  Declare Xtal = 16
  Dim MyWord1 as Word
  Dim MyWord2 as Word

  MyWord1 = 1575
  MyWord2 = 976
  MyWord1 = MyWord1 + MyWord2    ' Add the numbers.
  Hrsout Dec MyWord1, 13         ' Display the result

' 32-bit addition
  Device = 24FJ64GA002
  Declare Xtal = 16
  Dim MyWord as Word
  Dim MyDword as Dword

  MyWord = 1575
  MyDword = 9763647
  MyDword = MyDword + MyWord     ' Add the numbers.
  Hrsout Dec MyDword, 13         ' 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.

Operands
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.

```
  Device = 24FJ64GA002
  Declare Xtal = 16
  Dim MyWord1 as Word
  Dim MyWord2 as Word

  MyWord1 = 1000
  MyWord2 = 999
  MyWord1 = MyWord1 - MyWord2 ' Subtract the values.
  Hrsout Dec MyWord1          ' Display the result

' 32-bit subtraction
  Device = 24FJ64GA002
  Declare Xtal = 16
  Dim MyWord as Word
  Dim MyDword as Dword

  MyWord = 1575
  MyDword = 9763647
  MyDword = MyDword - MyWord   ' Subtract the values.
  Hrsout Dec MyDword, 13       ' Display the result

' 32-bit signed subtraction
  Device = 24FJ64GA002
  Declare Xtal = 16
  Dim MyDword1 as SDword
  Dim MyDword2 as SDword

  MyDword1 = 1575
  MyDword2 = 9763647
  MyDword1 = MyDword1 - MyDword2   ' Subtract the values.
  Hrsout Sdec MyDword1, 13         ' 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.

Operands
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, or 0 to 4294967295, as well as floating point.

```
  Device = 24FJ64GA002
  Declare Xtal = 16
  Dim MyWord1 as Word
  Dim MyWord2 as Word

  MyWord1 = 1000
  MyWord2 = 19
  MyWord1 = MyWord 1 * MyWord2    ' Multiply MyWord1 by MyWrd2.
  Hrsout Dec MyWord1, 13          ' Display the result

' 32-bit multiplication
  Device = 24FJ64GA002
  Declare Xtal = 16
  Dim MyWord as Word
  Dim MyDword as Dword

  MyWord = 100
  MyDword = 10000
  MyDword = MyDword * MyWord      ' Multiply the numbers.
  Hrsout Dec MyDword, 13          ' 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.

Operands
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.

```
Device = 24FJ64GA002
Declare Xtal = 16
Dim MyWord1 as Word
Dim MyWord2 as Word

MyWord1 = $FDE8
MyWord2 = MyWord1 ** MyWord1   ' Multiply $FDE8 by itself
Hrsout Hex MyWord2, 13         ' 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 Proton24 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.

Operands
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: -

```
Device = 24FJ64GA002
Declare Xtal = 16
Dim MyWord1 as Word

MyWord1 = 100
MyWord1 = MyWord1 */ $0180   ' Multiply by 1.5 [1 + (128/256)]
Hrsout Dec MyWord1, 13       ' 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: -

int(fraction * 256)

For example, take Pi (3.14159). The upper byte would be $03 (the whole number), and the lower would be 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, to some extent, melab's compiler code, but is rather obsolete considering the 32-bit capabilities of the Proton24 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.

Operands
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.

```
  Device = 24FJ64GA002
  Declare Xtal = 16
  Dim MyWord1 as Word
  Dim MyWord2 as Word

  MyWord1 = 1000
  MyWord2 = 5
  MyWord1 = MyWord1 / MyWord2  ' Divide the numbers.
  Hrsout Dec MyWord1, 13       ' Display the result (200).

' 32-bit division
  Device = 24FJ64GA002
  Declare Xtal = 16
  Dim MyWord as Word
  Dim MyDword as Dword

  MyWord = 100
  MyDword = 10000
  MyDword = MyDword / MyWord   ' Divide the numbers.
  Hrsout Dec MyDword, 13       ' Display the result
```

Note
The PIC24® and dsPIC33® range of devices have an exception mechanism built into their hardware. One of these mechanisms is a trap of a division by zero on any of its instructions. The compiler attempts to avoid this state, however, it cannot guarantee that an exception will not occur, and because it is a compiled language, it cannot always give an error message for such an event.

If the BASIC code seems to reset at a particular place within the code, make sure it is not a division by 0. If a suspected division by zero will occur, wrap the division within a condition. For example:

```
If MyVar2 <> 0 Then
  DestVar = MyVar1 / MyVar2
EndIF
```


Remainder '//'

Syntax
Assignment Variable = *Variable* // *Variable*

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

Operands
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: -

```
  Device = 24FJ64GA002
  Declare Xtal = 16
  Dim MyWord1 as Word
  Dim MyWord2 as Word

  MyWord1 = 1000
  MyWord2 = 6
  MyWord1 = MyWord1 // MyWord2   ' Get remainder of MyWord1 / MyWord2.
  Hrsout Dec MyWord1, 13         ' Display the result (4).

' 32-bit modulus
  Device = 24FJ64GA002
  Declare Xtal = 16
  Dim MyWord as Word
  Dim MyDword as Dword

  MyWord = 100
  MyDword = 99999
  MyDword = MyDword // MyWord    ' Mod the numbers.
  Hrsout Dec MyDword, 13         ' Display the result
```

The modulus operator does not operate with floating point values or variables. Use **fMod** for that operation.

Note
The PIC24® and dsPIC33® range of devices have an exception mechanism built into their hardware. One of these mechanisms is a trap of a division by zero on any of its instructions. The compiler attempts to avoid this state, however, it cannot guarantee that an exception will not occur, and because it is a compiled language, it cannot always give an error message for such an event.

If the BASIC code seems to reset at a particular place within the code, make sure it is not a division by 0. If a suspected division by zero will occur, wrap the modulus within a condition. For example:

```
If MyVar2 <> 0 Then
  DestVar = MyVar1 // MyVar2
EndIF
```

Logical and '&'

The And operator (&) returns the bitwise and of two integer 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: -

```
Device = 24FJ64GA002
Declare Xtal = 16
Dim MyByte1 as Byte
Dim MyByte2 as Byte
Dim Result as Byte
MyByte1 = %00001111
MyByte2 = %10101101
Result = MyByte1 & MyByte2
Hrsout Bin Result, 13       ' Display and result (%00001101)
```

or

```
Hrsout Bin (%00001111 & %10101101) , 13   ' 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 integer 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: -

```
Device = 24FJ64GA002
Declare Xtal = 16
Dim MyByte1 as Byte
Dim MyByte2 as Byte
Dim Result as Byte
MyByte1 = %00001111
MyByte2 = %10101001
Result = MyByte1 | MyByte2
Hrsout Bin Result, 13     ' Display or result (%10101111)
```

or

```
Hrsout Bin (%00001111 | %10101001) , 13     ' 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 integer 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: -

```
Device = 24FJ64GA002
Declare Xtal = 16
Dim MyByte1 as Byte
Dim MyByte2 as Byte
Dim Result as Byte
MyByte1 = %00001111
MyByte2 = %10101001
Result = MyByte1 ^ MyByte2
Hrsout Bin Result, 13           ' Display xor result (%10100110)
```

or

```
Hrsout Bin (%00001111 ^ %10101001) , 13   ' Display xor result (%10100110)
```

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

Bitwise Shift Left '<<'

Shifts the bits of an integer 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 *nth* power.

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

```
Device = 24FJ64GA002
Declare Xtal = 16
Dim MyWord as Word
Dim Loop as Byte
MyWord = %1111111111111111
For Loop = 1 to 16                ' Repeat with loop = 1 to 16.
  Hrsout Bin MyWord << Loop, 13   ' Shift MyWord 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 an integer value 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 *nth* power.

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

```
Device = 24FJ64GA002
Declare Xtal = 16
Dim MyWord as Word
Dim Loop as Byte

MyWord = %1111111111111111
For Loop = 1 to 16                ' Repeat with loop = 1 to 16.
  Hrsout Bin MyWord >> Loop, 13   ' Shift MyWord right Loop places.
Next
```

Complement '~'

The Complement operator (~) inverts the bits of an integer 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 "bit-wise not".

```
Device = 24FJ64GA002
Declare Xtal = 16
Dim MyWord1 as Word
Dim MyWord2 as Word

MyWord2 = %1111000011110000
MyWord1 = ~ MyWord2             ' Complement MyWord2.
Hrsout Bin16 MyWord1, 13        ' Display the result
```

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

Abs

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

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

Operands
Assignment Variable can be any valid variable type.
Variable can be a constant, variable or expression.

32-bit Example
```
Device = 24FJ64GA002
Declare Xtal = 16

Dim MyDword1 as Dword      ' Declare an unsigned Dword variable
Dim MyDword2 as Dword      ' Declare an unsigned Dword variable

MyDword1 = -1234567        ' Load MyDword1 with value -1234567
MyDword2 = Abs(MyDword1)   ' Extract the absolute value from MyDword1
Hrsout Dec MyDword2, 13    ' Display the result, which is 1234567
```

32-bit Floating Point example
```
Device = 24FJ64GA002
Declare Xtal = 16

Dim MyFloat1 as Float      ' Declare a Float variable
Dim MyFloat2 as Float      ' Declare a Float variable

MyFloat1 = -12345          ' Load MyFloat1 with value -12345
MyFloat2 = Abs(MyFloat1)   ' Extract the absolute value from MyFloat1
Hrsout Dec MyFloat2, 13    ' Display the result, which is 12345
```

64-bit Floating Point example
```
Device = 24FJ64GA002
Declare Xtal = 16

Dim MyDouble1 as Double      ' Declare a Double variable
Dim MyDouble2 as Double      ' Declare a Double variable

MyDouble1 = -12345           ' Load MyDouble1 with value -12345
MyDouble2 = Abs(MyDouble1)   ' Extract the absolute value from MyDouble1
Hrsout Dec MyDouble2, 13     ' Display the result, which is 12345
```

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)) + (ISin(MyVar2))
```


fAbs

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

Overview
Return the absolute value of a constant, variable or expression as 32-bit floating point.

Operands
Assignment Variable can be any valid variable type.
Variable can be a constant, variable or expression.

Example
```
Device = 24FJ64GA002
Declare Xtal = 16

Dim MyFloat as Float        ' Declare a Float variable
Dim Floatout as Float       ' Declare a Float variable

MyFloat = -3.14             ' Load MyFloat with value -3.14
Floatout = fAbs(MyFloat)    ' Extract the absolute value from MyFloat
Hrsout Dec Floatout, 13     ' Display the result, which is 3.14
```

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)) + (Sin(MyVar2))
```


dAbs

Syntax
Assignment Variable = **dAbs**(*Variable*)

Overview
Return the absolute value of a constant, variable or expression as 64-bit floating point.

Operands
Assignment Variable can be any valid variable type.
Variable can be a constant, variable or expression.

Example
```
Device = 24FJ64GA002
Declare Xtal = 16

Dim MyDouble as Double          ' Declare a Double variable
Dim Doubleout as Double         ' Declare a Double variable

MyDouble = -3.14                ' Load My Double with value -3.14
Doubleout = dAbs(MyDouble)      ' Extract the absolute value from My Double
Hrsout Dec Doubleout, 13        ' Display the result, which is 3.14
```

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 = (dAbs(MyVar1)) + (dSin(MyVar2))
```


Acos

Syntax
Assignment Variable = **Acos**(*Variable*)

Overview
Deduce the Arc Cosine of a 32-bit floating point value

Operands
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
```
Device = 24FJ64GA002
Declare Xtal = 16
Dim Floatin as Float       ' Holds the value to Acos
Dim Floatout as Float      ' Holds the result of the Acos

Floatin = 0.8              ' Load the variable
Floatout = Acos(Floatin)   ' Extract the Acos of the value
Hrsout Dec Floatout, 13    ' Display the result
```

Notes.
Floating point trigonometry is rather 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)) + (Sin(MyVar2))
```


dAcos

Syntax
Assignment Variable = **dAcos**(*Variable*)

Overview
Deduce the Arc Cosine of a 64-bit floating point value

Operands
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 64-bit floating point **dAcos** is in radians. The value must be in the range of -1 to +1

Example
```
Device = 24FJ64GA002
Declare Xtal = 16
Dim Doublein as Double        ' Holds the value to Acos
Dim Doubleout as Double       ' Holds the result of the Acos

Doublein = 0.8                ' Load the variable
Doubleout = dAcos(Doublein)   ' Extract the Acos of the value
Hrsout Dec Doubleout, 13      ' Display the result
```

Notes.
64-bit floating point trigonometry is very 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 = (dAcos(MyVar1)) + (dSin(MyVar2))
```


Asin

Syntax
Assignment Variable = **Asin**(*Variable*)

Overview
Deduce the Arc Sine of a 32-bit floating point value

Operands
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
```
Device = 24FJ64GA002
Declare Xtal = 16
Dim Floatin as Float        ' Holds the value to Asin
Dim Floatout as Float       ' Holds the result of the Asin

Floatin = 0.8               ' Load the variable
Floatout = Asin(Floatin)    ' Extract the Asin of the value
Hrsout Dec Floatout, 13     ' Display the result
```

Notes.
Floating point trigonometry is rather 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 compared to integer maths.

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)) + (Sin(MyVar2))
```


dAsin

Syntax
Assignment Variable = **dAsin**(*Variable*)

Overview
Deduce the Arc Sine of a 64-bit floating point value

Operands
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 **dAsin** is in radians. The value must be in the range of -1 to +1

Example

```
Device = 24FJ64GA002
Declare Xtal = 16
Dim Doublein as Double         ' Holds the value to Asin
Dim Doubleout as Double        ' Holds the result of the Asin

Doublein = 0.8                 ' Load the variable
Doubleout = dAsin(Doublein)   ' Extract the Asin of the value
Hrsout Dec Doubleout, 13       ' Display the result
```

Notes.
64-bit floating point trigonometry is very 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 compared to integer maths.

The compiler uses the stack as temporary storage when performing 64-bit floating point library routines, therefore, it may be required to increase the stack size from the default 120 words to 200 words using the **Stack_Size** declare. If the program resets with a stack underflow/overflow exception when running, the stack size is insufficient and requires increasing.

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 = (dAsin(MyVar1)) + (dSin(MyVar2))
```


Atan

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

Overview
Deduce the arc tangent of a 32-bit floating point value.

Operands
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
```
Device = 24FJ64GA002
Declare Xtal = 16
Dim Floatin as Float        ' Holds the value to Atan
Dim Floatout as Float       ' Holds the result of the Atan

Floatin = 1                 ' Load the variable
Floatout = Atan(Floatin)    ' Extract the Atan of the value
Hrsout Dec Floatout, 13     ' Display the result
```

Notes.
Floating point trigonometry is rather 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 = (dAtan(MyVar1)) + (dSin(MyVar2))
```

dAtan

Syntax
Assignment Variable = **dAtan**(*Variable*)

Overview
Deduce the arc tangent of a 64-bit floating point value.

Operands
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
```
Device = 24FJ64GA002
Declare Xtal = 16
Dim Doublein as Double       ' Holds the value to Atan
Dim Doubleout as Double      ' Holds the result of the Atan

Doublein = 1                 ' Load the variable
Doubleout = dAtan(Doublein)  ' Extract the Atan of the value
Hrsout Dec Doubleout, 13     ' Display the result
```

Notes.
64-bit floating point trigonometry is very 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.

The compiler uses the stack as temporary storage when performing 64-bit floating point library routines, therefore, it may be required to increase the stack size from the default 120 words to 200 words using the **Stack_Size** declare. If the program resets with a stack underflow/overflow exception when running, the stack size is insufficient and requires increasing.

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 = (dAtan(MyVar1)) + (dCos(MyVar2))
```

Atan2

Syntax
Assignment Variable = **Atan2**(*yVariable*, *xVariable*)

Overview
Deduce the arc tangent of 32-bit floating point y/x.

Operands
Assignment Variable can be any valid variable type.
y 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 **Atan2** is in radians.
x 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 **Atan2** is in radians.

Example
```
Device = 24FJ64GA002
Declare Xtal = 16
Dim Floatin1 as Float
Dim Floatin2 as Float

Dim Floatout as Float      ' Holds the result of Atan2

Floatin1 = 3
Floatin2 = 3.4

Floatout = Atan2(Floatin1, Floatin2) ' Extract the Atan2 of the values
Hrsout Dec Floatout, 13    ' Display the result
```

Notes.
Floating point trigonometry is rather 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 = (Atan2(MyVar1, MyVar2)) + (Tan(MyVar3))
```

dAtan2

Syntax
Assignment Variable = **dAtan2**(*yVariable*, *xVariable*)

Overview
Deduce the arc tangent of 64-bit floating point y/x.

Operands
Assignment Variable can be any valid variable type.
y Variable can be a constant, variable or expression that requires the arc tangent (Inverse Tangent) extracted. The value expected and returned by the 64-bit floating point **dAtan2** is in radians.
x Variable can be a constant, variable or expression that requires the arc tangent (Inverse Tangent) extracted. The value expected and returned by the 64-bit floating point **dAtan2** is in radians.

Example
```
Device = 24FJ64GA002
Declare Xtal = 16
Dim Doublein1 as Double
Dim Doublein2 as Double

Dim Doubleout as Double      ' Holds the result of Atan2

Doublein1 = 3
Doublein2 = 3.4

Doubleout = dAtan2(Doublein1, Doublein2)' Perfrom the Atan of the values
Hrsout Dec Doubleout, 13     ' Display the result
```

Notes.
64-bit floating point trigonometry is very 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.

The compiler uses the stack as temporary storage when performing 64-bit floating point library routines, therefore, it may be required to increase the stack size from the default 120 words to 200 words using the **Stack_Size** declare. If the program resets with a stack underflow/overflow exception when running, the stack size is insufficient and requires increasing.

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 = (dAtan2(MyVar1, MyVar2)) + (dSin(MyVar3))
```


Ceil

Syntax
Assignment Variable = **Ceil**(*Variable*)

Overview
Deduce the ceil of a 32-bit floating point value. Returns the smallest whole value greater than or equal to *Variable.*

Operands
Assignment Variable can be any valid variable type.
Variable can be a floating point constant, variable or expression.

Example
```
Device = 24FJ64GA002
Declare Xtal = 16
Dim Floatin as Float        ' Holds the value to Ceil
Dim Floatout as Float       ' Holds the result of the Ceil

Floatin = 3.8               ' Load the variable
Floatout = Ceil(Floatin)    ' Extract the ceil value
Hrsout Dec Floatout, 13     ' Display the result (4.0)
```

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 = (Ceil(MyVar1)) + (Sin(MyVar2))
```


dCeil

Syntax
Assignment Variable = **dCeil**(*Variable*)

Overview
Deduce the ceil of a 64-bit floating point value. Returns the smallest whole value greater than or equal to *Variable.*

Operands
Assignment Variable can be any valid variable type.
Variable can be a floating point constant, variable or expression.

Example
```
Device = 24FJ64GA002
Declare Xtal = 16
Dim Doublein as Double       ' Holds the value to Ceil
Dim Doubleout as Double      ' Holds the result of the Ceil

Doublein = 3.8               ' Load the variable
Doubleout = dCeil(Doublein)  ' Extract the ceil value
Hrsout Dec Doubleout, 13     ' Display the result (4.0)
```

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 = (dCeil(MyVar1)) + MyVar2
```


Cos

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

Overview
Deduce the Cosine of a 32-bit floating point value

Operands
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
```
Device = 24FJ64GA002
Declare Xtal = 16
Dim Floatin as Float         ' Holds the value to Cos with
Dim Floatout as Float        ' Holds the result of the Cos

Floatin = 123                ' Load the variable
Floatout = Cos(Floatin)      ' Extract the Cosine of the value
Hrsout Dec Floatout, 13      ' Display the result
```

Notes.
Floating point trigonometry is rather 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
```


dCos

Syntax
Assignment Variable = **dCos**(*Variable*)

Overview
Deduce the Cosine of a 64-bit floating point value

Operands
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 **dCos** is in radians.

Example

```
Device = 24FJ64GA002
Declare Xtal = 16
Dim Doublein as Double        ' Holds the value to Cos with
Dim Doubleout as Double       ' Holds the result of the Cos

Doublein = 123                ' Load the variable
Doubleout = dCos(Doublein)    ' Extract the Cosine of the value
Hrsout Dec Doubleout, 13      ' Display the result
```

Notes.
64-bit floating point trigonometry is very 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.

The compiler uses the stack as temporary storage when performing 64-bit floating point library routines, therefore, it may be required to increase the stack size from the default 120 words to 200 words using the **Stack_Size** declare. If the program resets with a stack underflow/overflow exception when running, the stack size is insufficient and requires increasing.

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 = (dCos(MyVar1)) + MyVar2
```

Dcd
2 n -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: -

```
Device = 24FJ64GA002
Declare Xtal = 16
Dim MyWord1 as Word

MyWord1= Dcd 12          ' Set bit 12.
Hrsout Bin16 MyWord1   ' Display result (%0001000000000000)
```

Dcd does not (as yet) support **Double**, **Float** or **Dword** type variables. Therefore the highest value obtainable is 65535.

Dig '?'
In this form, the **?** operator is compatible with the BASIC Stamp, and the melab's PICBASIC Pro compiler. **?** 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: -

```
Device = 24FJ64GA002
Declare Xtal = 16
Dim Loop as Byte
Dim MyWord1 as Word

MyWord1= 9742
Hrsout MyWord1 ? 2, 13      ' Display digit 2 (7)
For Loop = 0 to 4
  Hrsout MyWord1 ? Loop, 13 ' Display digits 0 through 4 of 9742.
Next
```

Note
Dig does not support **Double**, **Float** or **Dword** type variables.

Exp

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

Overview
Deduce the exponential function of a 32-bit floating point value. This is *e* to the power of *value* where *e* is the base of natural logarithms. **Exp** 1 is 2.7182818.

Operands
Assignment Variable can be any valid variable type.
Variable can be a constant, variable or expression.

Example
```
Device = 24FJ64GA002
Declare Xtal = 16
Dim Floatin as Float         ' Holds the value to Exp with
Dim Floatout as Float        ' Holds the result of the Exp

Floatin = 1                  ' Load the variable
Floatout = Exp(Floatin)      ' Extract the Exp of the value
Hrsout Dec Floatout, 13      ' Display the result
```

Notes.
Floating point trigonometry is rather 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
```

dExp

Syntax
Assignment Variable = **dExp**(*Variable*)

Overview
Deduce the exponential function of a 64-bit floating point value. This is *e* to the power of *value* where *e* is the base of natural logarithms. **dExp** 1 is 2.7182818.

Operands
Assignment Variable can be any valid variable type.
Variable can be a constant, variable or expression.

Example
```
Device = 24FJ64GA002
Declare Xtal = 16
Dim Doublein as Double     ' Holds the value to Exp with
Dim Doubleout as Double    ' Holds the result of the Exp

Doublein = 1               ' Load the variable
Doubleout = dExp(Doublein)' Extract the Exp of the value
Hrsout Dec Doubleout, 13   ' Display the result
```

Notes.
64-bit floating point trigonometry is very 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.

The compiler uses the stack as temporary storage when performing 64-bit floating point library routines, therefore, it may be required to increase the stack size from the default 120 words to 200 words using the **Stack_Size** declare. If the program resets with a stack underflow/overflow exception when running, the stack size is insufficient and requires increasing.

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 = (dExp(MyVar1)) + MyVar2
```


Floor

Syntax
Assignment Variable = **Floor**(*Variable*)

Overview
Deduce the floor of a 32-bit floating point value. Returns the largest whole value less than or equal to *Variable.*

Operands
Assignment Variable can be any valid variable type.
Variable can be a floating point constant, variable or expression.

Example
```
Device = 24FJ64GA002
Declare Xtal = 16
Dim Floatin as Float       ' Holds the value to Asin
Dim Floatout as Float      ' Holds the result of the Asin

Floatin = 3.8              ' Load the variable
Floatout = Floor(Floatin) ' Extract the floor value
Hrsout Dec Floatout, 13    ' Display the result (3.0)
```

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 = (Floor(MyVar1)) + MyVar2
```

dFloor

Syntax
Assignment Variable = **dFloor**(*Variable*)

Overview
Deduce the floor of a 64-bit floating point value. Returns the largest whole value less than or equal to *Variable.*

Operands
Assignment Variable can be any valid variable type.
Variable can be a floating point constant, variable or expression.

Example
```
Device = 24FJ64GA002
Declare Xtal = 16
Dim Doublein as Double       ' Holds the value to Asin
Dim Doubleout as Double      ' Holds the result of the Asin

Doublein = 3.8               ' Load the variable
Doubleout = dFloor(Doublein) ' Extract the floor value
Hrsout Dec Doubleout, 13     ' Display the result (3.0)
```

Notes.
The compiler uses the stack as temporary storage when performing 64-bit floating point library routines, therefore, it may be required to increase the stack size from the default 120 words to 200 words using the **Stack_Size** declare. If the program resets with a stack underflow/overflow exception when running, the stack size is insufficient and requires increasing.

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 = (dFloor(MyVar1)) + MyVar2
```


fRound

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

Overview
Round a 32-bit floating point value, variable or expression to the nearest whole number.

Operands
Assignment Variable can be any valid variable type.
Variable can be a constant, variable or expression.

Example 1
```
Device = 24FJ64GA002
Declare Xtal = 16
Dim Floatin as Float         ' Holds the value to round
Dim Dwordout as Dword        ' Holds the result of fRound

Floatin = 1.9                ' Load the variable
Dwordout = fRound(Floatin)   ' Round to the nearest whole value
Hrsout Dec Dwordout, 13      ' Display the integer result (which is 2)
```

Example 2
```
Device = 24FJ64GA002
Declare Xtal = 16
Dim Floatin as Float         ' Holds the value to round
Dim Dwordout as Dword        ' Holds the result of fRound

Floatin = 1.2                ' Load the variable
Dwordout = fRound(Floatin)   ' Round to the nearest whole value
Hrsout Dec Dwordout, 13      ' Display the integer result (which is 1)
```

Notes.
Floating point routines are rather 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
```


dRound

Syntax
Assignment Variable = **dRound**(*Variable*)

Overview
Round a 64-bit floating point value, variable or expression to the nearest whole number.

Operands
Assignment Variable can be any valid variable type.
Variable can be a constant, variable or expression.

Example 1
```
Device = 24FJ64GA002
Declare Xtal = 16
Dim Doublein as Double        ' Holds the value to round
Dim Dwordout as Dword         ' Holds the result of dRound

Doublein = 1.9                ' Load the variable
Dwordout = dRound(Doublein)  ' Round to the nearest whole value
Hrsout Dec Dwordout, 13       ' Display the integer result (which is 2)
```

Example 2
```
Device = 24FJ64GA002
Declare Xtal = 16
Dim Doublein as Double        ' Holds the value to round
Dim Dwordout as Dword         ' Holds the result of dRound

Doublein = 1.2                ' Load the variable
Dwordout = dRound(Doublein)  ' Round to the nearest whole value
Hrsout Dec Dwordout, 13       ' Display the integer result (which is 1)
```

Notes.
64-bit floating point routines are very 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.

The compiler uses the stack as temporary storage when performing 64-bit floating point library routines, therefore, it may be required to increase the stack size from the default 120 words to 200 words using the **Stack_Size** declare. If the program resets with a stack underflow/overflow exception when running, the stack size is insufficient and requires increasing.

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 = (dRound(MyVar1)) + MyVar2
```

ISin

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

Overview
Deduce the integer Sine of an integer value

Operands
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
```
Device = 24FJ64GA002
Declare Xtal = 16
Dim ByteIn as Byte          ' Holds the value to Sin
Dim ByteOut as Byte         ' Holds the result of the Sin

ByteIn = 123                ' Load the variable
ByteOut = ISin(ByteIn)      ' Extract the integer Sin of the value
Hrsout Dec ByteOut, 13      ' Display the result
```

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

Operands
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
```
Device = 24FJ64GA002
Declare Xtal = 16
Declare Hserial_Baud = 9600      ' USART1 baud rate
Declare Hrsout1_Pin = PORTB.14  ' Select the pin for TX with USART1

Dim ByteIn as Byte         ' Holds the value to Cos
Dim ByteOut as Byte        ' Holds the result of the Cos

RPOR7 = 3                  ' Make PPS Pin RP14 U1TX
ByteIn = 123               ' Load the variable
ByteOut = ICos(ByteIn)     ' Extract the integer Cosine of the value
Hrsout Dec ByteOut, 13     ' Display the result
```

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

Operands
Assignment Variable can be any valid variable type.
Variable can be a constant, variable or expression that requires the Square Root extracted.

Example
```
Device = 24FJ64GA002
Declare Xtal = 16
Declare Hserial_Baud = 9600     ' USART1 baud rate
Declare Hrsout1_Pin = PORTB.14  ' Select the pin for TX with USART1

Dim ByteIn as Byte          ' Holds the value to Cos
Dim ByteOut as Byte         ' Holds the result of the Cos

RPOR7 = 3                   ' Make PPS Pin RP14 U1TX

ByteIn = 123                ' Load the variable
ByteOut = ISqr(ByteIn)      ' Extract the integer square root of the value
Hrsout Dec ByteOut, 13      ' Display the result
```

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 32-bit floating point value

Operands
Assignment Variable can be any valid variable type.
Variable can be a constant, variable or expression that requires the natural logarithm extracted.

Example
```
Device = 24FJ64GA002
Declare Xtal = 16
Declare Hserial_Baud = 9600      ' USART1 baud rate
Declare Hrsout1_Pin = PORTB.14   ' Select the pin for TX with USART1

Dim Floatin as Float          ' Holds the value to Log with
Dim Floatout as Float         ' Holds the result of the Log

RPOR7 = 3                     ' Make PPS Pin RP14 U1TX

Floatin = 1                   ' Load the variable
Floatout = Log(Floatin)       ' Extract the Log of the value
Hrsout Dec Floatout, 13       ' Display the result
```

Notes.
Floating point trigonometry is rather 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
```


dLog

Syntax
Assignment Variable = **dLog**(*Variable*)

Overview
Deduce the Natural Logarithm a 64-bit floating point value

Operands
Assignment Variable can be any valid variable type.
Variable can be a constant, variable or expression that requires the natural logarithm extracted.

Example
```
Device = 24FJ64GA002
Declare Xtal = 16
Declare Hserial_Baud = 9600     ' USART1 baud rate
Declare Hrsout1_Pin = PORTB.14  ' Select the pin for TX with USART1

Dim Doublein as Double        ' Holds the value to Log with
Dim Doubleout as Double       ' Holds the result of the Log

RPOR7 = 3                     ' Make PPS Pin RP14 U1TX

Doublein = 1                  ' Load the variable
Doubleout = dLog(Doublein)    ' Extract the Log of the value
Hrsout Dec Doubleout, 13      ' Display the result
```

Notes.
64-bit floating point trigonometry is very 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.

The compiler uses the stack as temporary storage when performing 64-bit floating point library routines, therefore, it may be required to increase the stack size from the default 120 words to 200 words using the **Stack_Size** declare. If the program resets with a stack underflow/overflow exception when running, the stack size is insufficient and requires increasing.

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 = (dLog(MyVar1)) + MyVar2
```


Log10

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

Overview
Deduce the Logarithm a 32-bit floating point value

Operands
Assignment Variable can be any valid variable type.
Variable can be a constant, variable or expression that requires the Logarithm extracted.

Example
```
Device = 24FJ64GA002
Declare Xtal = 16
Declare Hserial_Baud = 9600     ' USART1 baud rate
Declare Hrsout1_Pin = PORTB.14  ' Select the pin for TX with USART1

Dim Floatin as Float       ' Holds the value to Log10 with
Dim Floatout as Float      ' Holds the result of the Log10

RPOR7 = 3                  ' Make PPS Pin RP14 U1TX

Floatin = 1                ' Load the variable
Floatout = Log10(Floatin)  ' Extract the Log10 of the value
Hrsout Dec Floatout, 13    ' Display the result
```

Notes.
Floating point trigonometry is rather 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
```

dLog10

Syntax
Assignment Variable =**dL**(*Variable*)

Overview
Deduce the Logarithm a 64-bit floating point value

Operands
Assignment Variable can be any valid variable type.
Variable can be a constant, variable or expression that requires the Logarithm extracted.

Example
```
Device = 24FJ64GA002
Declare Xtal = 16
Declare Hserial_Baud = 9600      ' USART1 baud rate
Declare Hrsout1_Pin = PORTB.14  ' Select the pin for TX with USART1

Dim Doublein as Double        ' Holds the value to Log10 with
Dim Doubleout as Double       ' Holds the result of the Log10

RPOR7 = 3                     ' Make PPS Pin RP14 U1TX

Doublein = 1                  ' Load the variable
Doubleout = dLog10(Doublein) ' Extract the Log10 of the value
Hrsout Dec Doubleout, 13      ' Display the result
```

Notes.
64-bit floating point trigonometry is rather 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.

The compiler uses the stack as temporary storage when performing 64-bit floating point library routines, therefore, it may be required to increase the stack size from the default 120 words to 200 words using the **Stack_Size** declare. If the program resets with a stack underflow/overflow exception when running, the stack size is insufficient and requires increasing.

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 = (dLog10(MyVar1)) + MyVar2
```

Modf

Syntax
Assignment Variable = **Modf (***pVariable*, p*Whole*)

Overview
Split a 32-bit floating point value into fractional and whole parts.

Operands
Assignment Variable can be any valid variable type.
pVariable can be a floating point constant, variable or expression that will be split.
pWhole must be a 32-bit floating point variable that will hold the whole part of the split value.

Example
```
Device = 24FJ64GA002
Declare Xtal = 16

Dim Floatin as Float
Dim Whole as Float          ' Holds the whole part of the value
Dim Fraction as Float       ' Holds the fractional part of the value
Dim MyDword as Dword

Floatin = 3.14

Fraction = Modf(Floatin, Whole)  ' Split the value
MyDword = Whole                  ' Convert the whole part to an integer
Hrsout "Whole = " Dec MyDword, 13
Hrsout "Fraction = ", Dec Fraction, 13
```

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 = (Modf(MyVar1)) + MyVar2
```


Modd

Syntax
Assignment Variable = **Modd (***pVariable*, p*Whole*)

Overview
Split a 64-bit floating point value into fractional and whole parts.

Operands
Assignment Variable can be any valid variable type.
pVariable can be a floating point constant, variable or expression that will be split.
pWhole must be a 64-bit floating point variable that will hold the whole part of the split value.

Example
```
Device = 24FJ64GA002
Declare Xtal = 16

Dim Doublein as Double
Dim Whole as Double             ' Holds the whole part of the value
Dim Fraction as Double          ' Holds the fractional part of the value
Dim MyDword as Dword

Doublein = 3.14

Fraction = Modd(Doublein, Whole) ' Split the value
MyDword = Whole                  ' Convert the whole part to an integer
Hrsout "Whole = " Dec MyDword, 13
Hrsout "Fraction = ", Dec Fraction, 13
```

Notes.
The compiler uses the stack as temporary storage when performing 64-bit floating point library routines, therefore, it may be required to increase the stack size from the default 120 words to 200 words using the **Stack_Size** declare. If the program resets with a stack underflow/overflow exception when running, the stack size is insufficient and requires increasing.

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 = (Modd(MyVar1)) + MyVar2
```

Ncd

Priority encoder of a 16-bit or 32-bit value. Ncd takes a value, finds the highest bit containing a 1 and returns the bit position plus one (1 through 32). 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: -

```
MyWord1= %1101                ' Highest bit set is bit 3.
Hrsout Dec Ncd MyWord1, 13    ' Display the Ncd of MyWord1(4).
```

Ncd does not support **Float** or **Double** type variables.

Pow

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

Overview
Computes 32-bit Floating point *Variable* to the power of *Pow Variable*.

Operands
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
```
Device = 24FJ64GA002
Declare Xtal = 16
Declare Hserial_Baud = 9600     ' USART1 baud rate
Declare Hrsout1_Pin = PORTB.14  ' Select the pin for TX with USART1

Dim PowOf as Float
Dim Floatin as Float           ' Holds the value to Pow with
Dim Floatout as Float          ' Holds the result of the Pow

RPOR7 = 3                      ' Make PPS Pin RP14 U1TX

PowOf= 10
Floatin = 2                    ' Load the variable
Floatout = Pow(Floatin,PowOf)  ' Extract the Pow of the value
Hrsout Dec Floatout, 13        ' Display the result
```

Notes.
Floating point trigonometry is rather 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.

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 = (Pow(MyVar1)) + MyVar2
```

dPow

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

Overview
Computes 64-bit Floating point *Variable* to the power of *Pow Variable*.

Operands
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
```
Device = 24FJ64GA002
Declare Xtal = 16
Declare Hserial_Baud = 9600      ' USART1 baud rate
Declare Hrsout1_Pin = PORTB.14   ' Select the pin for TX with USART1

Dim PowOf as Double
Dim Doublein as Double            ' Holds the value to Pow with
Dim Doubleout as Double           ' Holds the result of the Pow

RPOR7 = 3                         ' Make PPS Pin RP14 U1TX

PowOf= 10
Doublein = 2                      ' Load the variable
Doubleout = Pow(Doublein,PowOf)  ' Extract the Pow of the value
Hrsout Dec Doubleout, 13          ' Display the result
```

Notes.
64-bit loating point trigonometry is rather 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.

The compiler uses the stack as temporary storage when performing 64-bit floating point library routines, therefore, it may be required to increase the stack size from the default 120 words to 200 words using the **Stack_Size** declare. If the program resets with a stack underflow/overflow exception when running, the stack size is insufficient and requires increasing.

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 = (dPow(MyVar1)) + MyVar2
```

Rev '@'

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

```
  Var1 = %10101100 @ 4  ' Sets Var1 to %10100011
```

or

```
  Dim MyDword as Dword
' Sets MyDword to %10101010000000001111111110100011
  MyDword = %10101010000000001111111110101100 @ 4
```

Sin

Syntax
Assignment Variable = **Sin**(*Variable*)

Overview
Deduce the Sine of a 32-bit floating point value

Operands
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
```
Device = 24FJ64GA002
Declare Xtal = 16
Declare Hserial_Baud = 9600      ' USART1 baud rate
Declare Hrsout1_Pin = PORTB.14   ' Select the pin for TX with USART1

Dim Floatin as Float        ' Holds the value to Sin
Dim Floatout as Float       ' Holds the result of the Sin

RPOR7 = 3                   ' Make PPS Pin RP14 U1TX

Floatin = 123               ' Load the variable
Floatout = Sin(Floatin)     ' Extract the Sin of the value
Hrsout Dec Floatout, 13     ' Display the result
```

Notes.
Floating point trigonometry is rather 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
```

dSin

Syntax
Assignment Variable = **dSin**(*Variable*)

Overview
Deduce the Sine of a 64-bit floating point value

Operands
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 **dSin** is in radians.

Example
```
Device = 24FJ64GA002
Declare Xtal = 16
Declare Hserial_Baud = 9600     ' USART1 baud rate
Declare Hrsout1_Pin = PORTB.14  ' Select the pin for TX with USART1

Dim Doublein as Double         ' Holds the value to Sin
Dim Doubleout as Double        ' Holds the result of the Sin

RPOR7 = 3                      ' Make PPS Pin RP14 U1TX

Doublein = 123                 ' Load the variable
Doubleout = dSin(Doublein)     ' Extract the Sin of the value
Hrsout Dec Doubleout, 13       ' Display the result
```

Notes.
64-bit floating point trigonometry is rather 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.

The compiler uses the stack as temporary storage when performing 64-bit floating point library routines, therefore, it may be required to increase the stack size from the default 120 words to 200 words using the **Stack_Size** declare. If the program resets with a stack underflow/overflow exception when running, the stack size is insufficient and requires increasing.

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 = (dSin(MyVar1)) + MyVar2
```

Sqr

Syntax
Assignment Variable = **Sqr**(*Variable*)

Overview
Deduce the Square Root of a 32-bit floating point value

Operands
Assignment Variable can be any valid variable type.
Variable can be a constant, variable or expression that requires the Square Root extracted.

Example
```
Device = 24FJ64GA002
Declare Xtal = 16
Declare Hserial_Baud = 9600      ' USART1 baud rate
Declare Hrsout1_Pin = PORTB.14   ' Select the pin for TX with USART1

Dim Floatin as Float        ' Holds the value to Sqr
Dim Floatout as Float       ' Holds the result of the Sqr

RPOR7 = 3                   ' Make PPS Pin RP14 U1TX

Floatin = 600               ' Load the variable
Floatout = Sqr(Floatin)     ' Extract the Sqr of the value
Hrsout Dec Floatout, 13     ' Display the result
```

Notes.
Floating point trigonometry is rather 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
```


dSqr

Syntax
Assignment Variable = **dSqr**(*Variable*)

Overview
Deduce the Square Root of a 64-bit floating point value

Operands
Assignment Variable can be any valid variable type.
Variable can be a constant, variable or expression that requires the Square Root extracted.

Example
```
Device = 24FJ64GA002
Declare Xtal = 16
Declare Hserial_Baud = 9600      ' USART1 baud rate
Declare Hrsout1_Pin = PORTB.14   ' Select the pin for TX with USART1

Dim Doublein as Double          ' Holds the value to Sqr
Dim Doubleout as Double         ' Holds the result of the Sqr

RPOR7 = 3                       ' Make PPS Pin RP14 U1TX

Doublein = 600                  ' Load the variable
Doubleout = dSqr(Doublein)      ' Extract the Sqr of the value
Hrsout Dec Doubleout, 13        ' Display the result
```

Notes.
Floating point trigonometry is rather 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.

The compiler uses the stack as temporary storage when performing 64-bit floating point library routines, therefore, it may be required to increase the stack size from the default 120 words to 200 words using the **Stack_Size** declare. If the program resets with a stack underflow/overflow exception when running, the stack size is insufficient and requires increasing.

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 = (dSqr(MyVar1)) + MyVar2
```

Tan

Syntax
Assignment Variable = **Tan**(*Variable*)

Overview
Deduce the Tangent of a 32-bit floating point value

Operands
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
```
Device = 24FJ64GA002
Declare Xtal = 16
Declare Hserial_Baud = 9600      ' USART1 baud rate
Declare Hrsout1_Pin = PORTB.14   ' Select the pin for TX with USART1

Dim Floatin as Float        ' Holds the value to Tan
Dim Floatout as Float       ' Holds the result of the Tan

RPOR7 = 3                   ' Make PPS Pin RP14 U1TX

Floatin = 1                 ' Load the variable
Floatout = Tan(Floatin)     ' Extract the Tan of the value
Hrsout Dec Floatout, 13     ' Display the result
```

Notes.
Floating point trigonometry is rather 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
```

dTan

Syntax
Assignment Variable = **dTan**(*Variable*)

Overview
Deduce the Tangent of a 64-bit floating point value

Operands
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 **dTan** is in radians.

Example
```
Device = 24FJ64GA002
Declare Xtal = 16
Declare Hserial_Baud = 9600     ' USART1 baud rate
Declare Hrsout1_Pin = PORTB.14  ' Select the pin for TX with USART1

Dim Doublein as Double        ' Holds the value to Tan
Dim Doubleout as Double       ' Holds the result of the Tan

RPOR7 = 3                     ' Make PPS Pin RP14 U1TX

Doublein = 1                  ' Load the variable
Doubleout = dTan(Doublein)    ' Extract the Tan of the value
Hrsout Dec Doubleout, 13      ' Display the result
```

Notes.
64-bit floating point trigonometry is very 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.

The compiler uses the stack as temporary storage when performing 64-bit floating point library routines, therefore, it may be required to increase the stack size from the default 120 words to 200 words using the **Stack_Size** declare. If the program resets with a stack underflow/overflow exception when running, the stack size is insufficient and requires increasing.

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 = (dTan(MyVar1)) + MyVar2
```

Commands and Directives

Adin Read the on-board analogue to digital converter.
AddressOf Locate the address of a variable or label.
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^2C bus.
Bstop Send a **Stop** condition to the I^2C bus.
Brestart Send a **Restart** condition to the I^2C bus.
BusAck Send an **Acknowledge** condition to the I^2C bus.
BusNack Send an **Not Acknowledge** condition to the I^2C bus.
Busin Read bytes from an I^2C device using a bit-bashed interface.
Busout Write bytes to an I^2C device using a bit-bashed interface.
Button Detect and debounce a key press.
Call Call an assembly language subroutine.
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 an 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.
cPtr8, cPtr16, cPtr32, cPtr64 Indirectly read code memory using a variable as the address.
Cread8, Cread16, Cread32, Cread64 Read a value from a code memory table.
Cursor Position the cursor on the LCD.
Dec Decrement a variable.
Declare Adjust library routine parameters.
DelayMs Delay (1 millisecond resolution).
DelayUs Delay (1 microsecond resolution).
DelayCs Delay (1 clock cycle resolution).
Device Choose the type of PIC24® or dsPIC33® microcontroller to compile for.
Dig Return the value of a decimal digit.
Dim Create a variable.
DTMFout Produce a DTMF Touch Tone note.
Edata Define initial contents of on-board eeprom.
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^2C bus using the MSSP module.
HbStop Send a **Stop** condition to the I^2C bus using the MSSP module.
HbRestart Send a **Restart** condition to the I^2C bus using the MSSP module.
HbusAck Send an **Ack** condition to the I^2C bus using the MSSP module.

HbusNack Send a **Not Ack** condition to the I^2C bus using the MSSP module.
Hbusin Read from an I^2C device using the MSSP module.
Hbusout Write to an I^2C 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.
Hrsin3 Same as **Hrsin** but using a 3rd USART if available.
Hrsout3 Same as **Hrsout** but using a 3rd USART if available.
Hserin3 Same as **Hserin** but using a 3rd USART if available.
Hserout3 Same as **Hserout** but using a 3rd USART if available.
Hrsin4 Same as **Hrsin** but using a 4th USART if available.
Hrsout4 Same as **Hrsout** but using a 4th USART if available.
Hserin4 Same as **Hserin** but using a 4th USART if available.
Hserout4 Same as **Hserout** but using a 4th USART if available.
I2Cin Read bytes from an I^2C device with user definable SDA\SCL lines.
I2Cout Write bytes to an I^2C 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.
ISR_Start...ISR_End Define an interrupt handler block of code.
LCDread Read a single byte from a Graphic LCD.
LCDwrite Write bytes to a Graphic LCD.
Left$ Extract *n* amount of characters from the left of a String.
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.
Mid$ Extract characters from a String beginning at *n* characters from the left.
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.
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.
Ptr8, Ptr16, Ptr32, Ptr64 Indirectly read or write RAM using a variable to hold the address.
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.
Return Continue at the statement following the last **Gosub**.
Right$ Extract *n* amount of characters from the right of a String.
Rol Bitwise rotate a variable left with or without the microcontroller's Carry flag.
Ror Bitwise rotate a variable right with or without the microcontroller's Carry flag.
Rsin Asynchronous serial input from a fixed pin and baud rate.
Rsout 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.
SetBit Set a bit of a port or variable, using a variable index.
Shin Synchronous serial input.
Shout Synchronous serial output.
Sound Generate a tone or white-noise on a specified pin.
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.
Touch_Active Detect if the touch screen is being touched.
Touch_Read Read the X and Y coordinates from the touch screen
Touch_HotSpot Detect a touch within a user defind window on the touch screen
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.
Val Convert a null terminated String to an integer value.
While…Wend Execute statements while condition is true.

Adin

Syntax
Variable **= Adin** *AN number*

Overview
Read the value from the on-board Analogue to Digital Converter.

Operands
Variable is a user defined variable that holds the result of the ADC.
Ad number can be a constant, variable or expression. It holds the ADC number from MUXA of the device in use.

Example
```
' Read the value from AN0 of the ADC and serially transmit the result.
  Device = 24FJ64GA002
  Declare Xtal = 20

  Declare Adin_Tad = cFRC         ' RC oscillator chosen
  Declare Adin_Stime = 10         ' Allow 10us sample time

  Declare Hserial_Baud = 9600     ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14  ' Pin to be used for TX with USART1

  Dim MyWord as Word          ' Create a word variable

  RPOR7 = 3                   ' Make PPS Pin RP14 U1TX
  AD1CON2 = 0                 ' AVdd, AVss, MUXA only
  AD1PCFGbits_PCFG0 = 0       ' Analogue input on AN0
  While                       ' Create an infinite loop
  MyWord = Adin 0             ' Place the conversion into variable MyWord
    Hrsout Dec MyWord, 13     ' Transmit the decimal ASCII result
    DelayUs 2                 ' Wait for 2us
  Wend                        ' do it forever
```

Adin Declares
There are two **Declare** directives for use with **Adin.** These are: -

Declare Adin_Tad c1_FOSC, **c2_FOSC**, **c4_FOSC**, **c8_FOSC**, **c16_FOSC**, **c32_FOSC**, **c64_FOSC**, or **cFRC**.
Sets the ADC's clock source.

All compatible devices have multiple options for the clock source used by the ADC peripheral. c1_FOSC, c2_FOSC, c4_FOSC, c8_FOSC, c16_FOSC, c32_FOSC, and c64_FOSC are ratios of the external oscillator, while FRC is the device's internal RC oscillator.

Care must be used when issuing this **Declare**, as the wrong type of clock source may result in poor accuracy, or no conversion at all. If in doubt use FRC which will produce a slight reduction in accuracy 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 2 to 100. This allows adequate charge time without loosing 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 pin must be configured as an analogue input. Refer to the datasheet of the specific device being used for more information on the SFRs involved.

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: -

```
While                ' Create an infinite loop
MyWord = Adin 0      ' Place the conversion into variable MyWord
  DelayUs 4          ' Wait for 4us
Wend                 ' Read the ADC forever
```

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.

The compiler also allows simple assembler mnemonics to be used within the BASIC program without wrapping them in **Asm**-**EndAsm**.

Box

Syntax
Box *Pixel Colour*, *Xpos Start*, *Ypos Start*, *Size*

Overview
Draw a square on a graphic LCD.

Operands
Pixel Colour 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. If using a colour graphic LCD, this parameter holds the 16-bit colour of the pixel.
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 65535.
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 65535.
Size may be a constant or variable that holds the Size of the square (in pixels). Can be a value from 0 to 65535.

KS0108 graphic LCD example
```
' Draw a square at position 63,32 with a size of 20 pixels
' on a Samsung KS0108 graphic LCD
    Device = 24FJ64GA002
    Declare Xtal = 16

    Declare LCD_Type = Samsung     ' Setup for a Samsung KS0108 graphic LCD
    Declare LCD_DTPort = PORTB.Byte0 ' Use the first 8-bits of PORTB
    Declare LCD_CS1Pin = PORTB.8
    Declare LCD_CS2Pin = PORTB.9
    Declare LCD_ENPin = PORTB.10
    Declare LCD_RSPin = PORTB.11
    Declare LCD_RWPin = PORTB.12

    Dim Xpos as Byte
    Dim Ypos as Byte
    Dim Size as Byte
    Dim SetClr as Byte

    DelayMs 100          ' Wait for things to stabilise
    Cls                  ' Clear the LCD
    Xpos = 63
    Ypos = 32
    Size = 20
    SetClr = 1
    Box SetClr, Xpos, Ypos, Radius
```

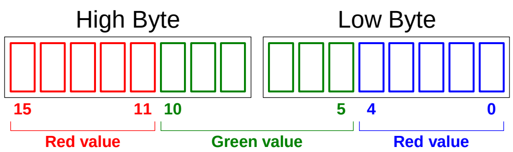

ILI9320 colour graphic LCD example

```
' Demonstrate the box command with a colour LCD
'
  Device = 24EP128MC202
  Declare Xtal = 140.03
'
' Setup the Pins used by the ILI9320 graphic LCD
'
  Declare LCD_DTPort = PORTB.Byte0     ' Use the first 8-bits of PORTB
  Declare LCD_CSPin = PORTB.8          ' Connect to the LCD's CS pin
  Declare LCD_RDPin = PORTB.9          ' Connect to the LCD's RD pin
  Declare LCD_RSPin = PORTB.10         ' Connect to the LCD's RS pin
  Declare LCD_WRPin = PORTA.3          ' Connect to the LCD's WR pin

  Include "ILI9320.inc"      ' Load the ILI9320 routines into the program

  Dim wRadius As Word        ' Create a variable for the circle's radius

'--------------------------------------------------------------------------
Main:
' Configure the internal Oscillator to operate the device at 140.03MHz
'
  PLL_Setup(76, 2, 2, $0300)

  Cls clYellow                 ' Clear the LCD with the colour yellow
  For wRadius = 0 To 319
    Box clBrightCyan, 120, 160, wRadius ' Draw a series of squares
  Next
'--------------------------------------------------------------------------
' Configure for internal 7.37MHz oscillator with PLL
' OSC pins are general purpose I/O
'
  Config FGS = GWRP_OFF, GCP_OFF
  Config FOSCSEL = FNOSC_FRCPLL, IESO_ON, PWMLOCK_OFF
  Config FOSC = POSCMD_NONE, OSCIOFNC_ON, IOL1WAY_OFF, FCKSM_CSDCMD
  Config FWDT = WDTPOST_PS256, WINDIS_OFF, PLLKEN_ON, FWDTEN_OFF
  Config FPOR = ALTI2C1_ON, ALTI2C2_OFF
  Config FICD = ICS_PGD1, JTAGEN_OFF
```

Notes.

Because of the aspect ratio of the pixels on the KS0108 graphic LCD (approx 1.5 times higher than wide) the circle will appear elongated.

With an ILI9320 colour graphic LCD, the colour is a 16-bit value formatted in RGB565, where the upper 5-bits represent the red content, the middle 6-bits represent the green content, and the lower 5-bits represent the blue content. As illustrated below:



For convenience, there are several colours defined within the ILI9320.inc file. These are:

```
clBlack
clBrightBlue
clBrightGreen
clBrightCyan
clBrightRed
clBrightMagenta
clBrightYellow
clBlue
clGreen
clCyan
clRed
clMagenta
clBrown
clLightGray
clDarkGray
clLightBlue
clLightGreen
clLightCyan
clLightRed
clLightMagenta
clYellow
clWhite
```

More constant values for colours can be added by the user if required.

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, where the destination is within 32768 bytes from the source.

Operands
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 65536 labels may be placed between the square brackets.

Example
```
  Device = 24FJ64GA002
  Declare Xtal = 16

  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 less code memory. If larger devices are used and you suspect that the branch label will be over the 32768 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.

Operands
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 65536 labels may be placed between the square brackets.

Example
```
  Device = 24FJ64GA002
  Declare Xtal = 16

  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].

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

  Device = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600     ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14  ' Select the pin for TX with USART1

  Dim MyByte as Byte

  RPOR7 = 3                      ' Make PPS Pin RP14 U1TX
  For MyByte = 0 to 39           ' Create a loop of 40 revolutions
    Hrsout Dec MyByte, 13        ' Display revolutions on the serial terminal
    If MyByte = 10 Then Break    ' Break out of the loop when MyByte = 10
    DelayMs 200                  ' Delay so we can see what's happening
  Next                           ' Close the For-Next loop
  Hrsout "Exited At ", Dec MyByte, 13 ' Display value when loop was broken
```

Example 2

```
' Break out of a Repeat-Until loop when the count reaches 10

  Device = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600     ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14  ' Select the pin for TX with USART1

  Dim MyByte as Byte

  RPOR7 = 3                      ' Make PPS Pin RP14 U1TX
  MyByte = 0
  Repeat                         ' Create a loop
    Hrsout Dec MyByte, 13        ' Display revolutions on the serial terminal
    If MyByte = 10 Then Break    ' Break out of the loop when MyByte = 10
    DelayMs 200                  ' Delay so we can see what's happening
    Inc MyByte
  Until MyByte > 39              ' Close the loop after 40 revolutions
  Hrsout "Exited At ", Dec MyByte, 13 ' Display value when loop was broken
```

Example 3

```
' Break out of a While-Wend loop when the count reaches 10

  Device = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600      ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14   ' Select the pin for TX with USART1

  Dim MyByte as Byte

  RPOR7 = 3                       ' Make PPS Pin RP14 U1TX
  MyByte = 0
  While MyByte < 40                ' Create a loop of 40 revolutions
    Hrsout Dec MyByte, 13          ' Display revolutions on the serial terminal
    If MyByte = 10 Then Break      ' Break out of the loop when MyByte = 10
    DelayMs 200                    ' Delay so we can see what's happening
    Inc MyByte
  Wend                             ' Close the loop
  Hrsout "Exited At ", Dec MyByte, 13 ' Display value when loop was broken
```

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.

See also : **Continue, For**…**Next, While**…**Wend, Repeat**…**Until.**

Bstart

Syntax
Bstart

Overview
Send a **Start** condition to the I^{2}C bus.

Notes.
Because of the subtleties involved in interfacing to some I^{2}C devices, the compiler's standard **Busin**, and **Busout** commands were found lacking somewhat. Therefore, individual pieces of the I^{2}C 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 = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600      ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14   ' Select the pin for TX with USART1
  Declare SCL_Pin = PORTB.3        ' Select the pin for I2C SCL
  Declare SDA_Pin = PORTB.4        ' Select the pin for I2C SDA

  Dim Loop as Byte
  Dim MyString as String * 20
  RPOR7 = 3                      ' Make PPS Pin RP14 U1TX
'
' 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 5               ' Wait for data to be entered into eeprom matrix
'
' Receive bytes from the I2C bus
'
  Clear MyString          ' Clear the string before we start
  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
    MyString[Loop] = Busin ' Load a string with bytes received
    If Loop = 9 Then Bstop : Else : BusAck ' Ack or Stop ?
  Next                    ' Close the loop
  Hrsout MyString, 13     ' Display the String
```

See also: **Bstop, Brestart, BusAck, Busin, Busout, HbStart, HbRestart, HbusAck, Hbusin, Hbusout.**

Bstop

Syntax
Bstop

Overview
Send a **Stop** condition to the I^{2}C bus.

Brestart

Syntax
Brestart

Overview
Send a **Restart** condition to the I^{2}C bus.

BusAck

Syntax
BusAck

Overview
Send an **Acknowledge** condition to the I^{2}C bus.

BusNack

Syntax
BusNack

Overview
Send a **Not Acknowledge** condition to the I^{2}C 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* out of the clock pin (*SCL*), and data pin (*SDA*).

Operands
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 is a software implementation (bit-bashed) and 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 MyByte.
  Device = 24FJ64GA002
  Declare Xtal = 16
  Declare SCL_Pin = PORTB.3        ' Select the pin for I2C SCL
  Declare SDA_Pin = PORTB.4        ' Select the pin for I2C SDA

  Dim Loop as Byte
  Dim MyString as String * 20
  Dim MyByte 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
  MyByte = Busin Control, Address  ' Read the byte from the eeprom
```

or

```
  Busin Control, Address, [ MyByte ] ' 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 MyWord as Word          ' Declare a Word size variable
  MyWord = Busin Control, Address
```

Will receive a 16-bit value from the bus. While: -

```
  Dim MyByte as Byte        ' Declare a Byte size variable
  MyByte = 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 MyByte as Byte
  Dim MyWord as Word
  Busin Control, Address, [MyByte, MyWord]
```

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 MyWord 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 I^{2}C 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 I^{2}C 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 I^{2}C 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 microcontroller 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 I^{2}C 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: -

```
  Device = 24FJ64GA002
  Declare Xtal = 16
  Declare SCL_Pin = PORTB.3         ' Select the pin for I2C SCL
  Declare SDA_Pin = PORTB.4         ' Select the pin for I2C SDA

  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 I^{2}C 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.

Operands
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 is a software implementation (bit-bashed) and operates as an I^{2}C master without using the device's MSSP module, and may be used to interface with any device that complies with the 2-wire I^{2}C 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.
  Device = 24FJ64GA002
  Declare Xtal = 16
  Declare SCL_Pin = PORTB.3          ' Select the pin for I2C SCL
  Declare SDA_Pin = PORTB.4          ' Select the pin for I2C SDA

  Dim MyByte 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
  MyByte = 200                       ' The value place into address 20
  Busout Control, Address,[MyByte] ' 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 MyWord as Word          ' Declare a Word size variable
Busout Control, Address, [MyWord]
```

Will send a 16-bit value to the bus. While: -

```
Dim MyByte as Byte          ' Declare a Byte size variable
Busout Control, Address, [MyByte]
```

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 MyByte as Byte
Dim MyWord as Word
Busout Control, Address, [MyByte, MyWord]
```

Will send two values to the bus, the first being an 8-bit value dictated by the size of variable MyByte which has been declared as a byte. And a 16-bit value, this time dictated by the size of the variable MyWord 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", MyByte, MyWord]
```

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 I^2C 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 microcontroller's Carry flag, which is SR.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
  Device = 24FJ64GA002
  Declare Xtal = 16
  Declare SCL_Pin = PORTB.3          ' Select the pin for I2C SCL
  Declare SDA_Pin = PORTB.4          ' Select the pin for I2C SDA

  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 SRbits_C = 1 Then GoTo Not_Received ' Has Ack been received OK?
  Bstop                     ' Send a Stop condition
  DelayMs 5                 ' 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: -

```
Device = 24FJ64GA002
Declare Xtal = 16
Declare SCL_Pin = PORTB.3          ' Select the pin for I2C SCL
Declare SDA_Pin = PORTB.4          ' Select the pin for I2C SDA

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: -

```
Device = 24FJ64GA002
Declare Xtal = 16
Declare SCL_Pin = PORTB.3          ' Select the pin for I2C SCL
Declare SDA_Pin = PORTB.4          ' Select the pin for I2C SDA
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 micro-controller.

Declare SCL_Pin Port **.** Pin
Declares the port and pin used for the clock line (SCL). This may be any valid port on the micro-controller.

These declares, as is the case with all the Declares, may only be issued once in any single program, as they setup the I^{2}C library code at design time.

Declare Slow_Bus On - Off or 1 - 0
Slows the bus speed when using an oscillator higher than 4MHz.

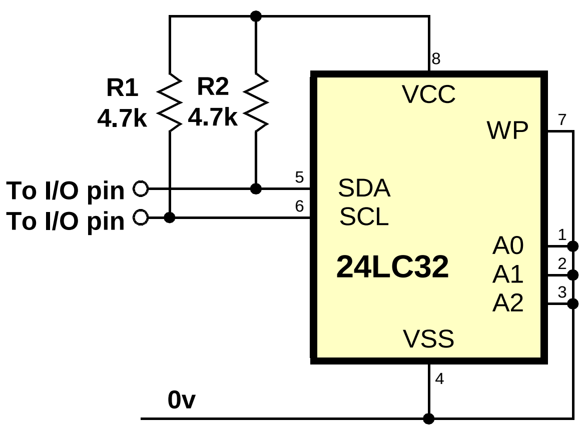
The standard speed for the I^{2}C bus is 100KHz. Some devices use a higher bus speed of 400KHz. If you use an 8MHz or higher oscillator, the bus speed may exceed the devices specs, which will result in intermittent transactions, or in some cases, no transactions at all. Therefore, use this **Declare** 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 I^{2}C 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 I^{2}C 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 device, in order to interface to many devices.

A typical use for the I^{2}C commands is for interfacing with serial eeproms. Shown below is the connections to the I^{2}C 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.

Operands
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
```
  Device = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600      ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14  ' Select the pin for TX with USART1

  Dim BtnVar as Byte               ' Workspace for Button instruction.

  RPOR7 = 3                       ' Make PPS Pin RP14 U1TX
Loop:
  ' Go to NoPress unless BtnVar = 0.
  Button 0, 0, 255, 250, BtnVar, 0, NoPress
  Hrsout "*\r"
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.

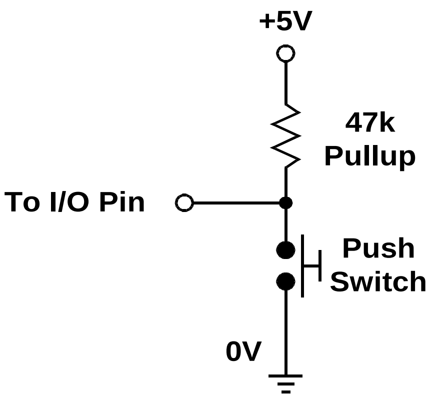
Button also reacts to a button press the way a computer keyboard does to a key press. When a key is pressed, a character immediately appears on the screen. If the key is held down, there's a delay, then a rapid stream of characters appears on the screen. **Button**'s auto-repeat function can be set up 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).

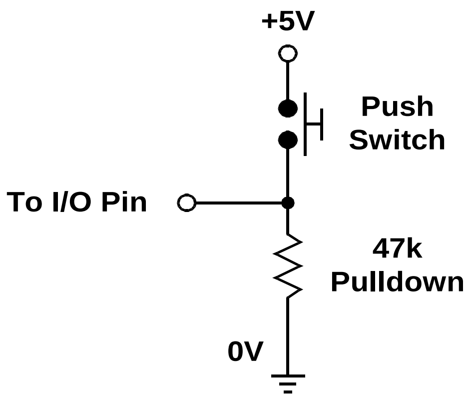
If the switch stays in *DownState,* **Button** counts the number of program loops that execute. When this count equals *Delay,* **Button** once again triggers the action specified by *TargetState* and *address.* Thereafter, if the switch remains in *DownState*, **Button** waits *Rate* number of cycles 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.



Active Low



Active High

Call

Syntax
Call *Label*

Overview
Execute the assembly language subroutine named *label*.

Operands
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 or linker types, which can be misleading and rather obscure.

See also : Gosub, GoTo

Cdata

Syntax
Cdata { *alphanumeric data* }

Overview
Place information directly into memory for access by **Cread8**, **Cread16**, **Cread32** and **Cread64**.

Operands
alphanumeric data can be any value, alphabetic character, or string enclosed in quotes (") or numeric data without quotes.

Example
```
  Device = 24FJ64GA002
  Declare Xtal = 16

  Dim MyByte as Byte
  Dim bIndex as Byte
  RPOR7 = 3                        ' Make PPS Pin RP14 U1TX
  For Loop = 0 to 11                ' Create a loop of 12
    MyByte = Cread8 MyLabel[bIndex]  ' Read memory location MyLabel + bIndex
    Hrsout MyByte                  ' Display the value read
  Next
  Stop
MyLabel:
  Cdata "Hello World\r"      ' Create a string of text in code memory
```

The program above reads and displays 12 values from the address located by the Label accompanying the **Cdata** command. Resulting in "Hello World" being displayed.

Formatting a Cdata 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.

```
  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 one of the **Cread** commands 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
Double

Placing one of these formatters at the beginning of the table will force a given length.

```
  Cdata  As Dword 100000, 10000, 1000, 100, 10, 1
```

Byte will force all values in the table 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 values in the table 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 values in the table 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 the values in the table to their floating point equivalent, which always takes up 4 bytes of code space.

The example below illustrates the formatters in use.

```
' Convert a Dword value into a string. Using only BASIC commands
' Similar principle to the Str$ command

  Device = 24FJ64GA002
  Declare Xtal = 16

  Dim Pow10 as Dword             ' Power of 10 variable
  Dim bCount as Byte
  Dim bTableIndex as Byte
  Dim dValue as Dword            ' Value to convert
  Dim MyString as String * 12  ' Holds the converted value
  Dim bIndex as Byte             ' Index within the string

  RPOR7 = 3                      ' Make PPS Pin RP14 U1TX
  DelayMs 10                     ' Wait for things to stabilise
  Clear                          ' Clear all RAM before we start
  Value = 1234576                ' Value to convert
  Gosub DwordToStr               ' Convert Value to string
  Hrsout MyString                ' Display the result
  Stop
' Convert a Dword value into a string array
' Value to convert is placed in 'Value'
' Byte array 'Array1' is built up with the ASCII equivalent
'
DwordToStr:
  bIndex = 0
  bTableIndex = 0
  Repeat
    Pow10 = Cread32 Dword_Table[bTableIndex]
    bCount = 0
    While dValue >= Pow10
      dValue = dValue - Pow10
      Inc bCount
    Wend

    If bCount <> 0 Then
      MyString [bIndex] = bCount + "0"
      Inc bIndex
    EndIf
    Inc bTableIndex
```

```
  Until bTableIndex > 8

  MyString[bIndex] = dValue + "0"
  Inc bIndex
  MyString[bIndex] = 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_Table:
  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.

```
' Transmit serially text from two code memory tables
' Based on their address located in a separate table

    Device = 24FJ64GA002
    Declare Xtal = 16
    Declare Hserial_Baud = 9600       ' USART1 baud rate
    Declare Hrsout1_Pin = PORTB.14    ' Select the pin used for TX with
USART1

    Dim MyByte As Byte
    Dim MyString1 As Code = "Hello",0
    Dim MyString2 As Code = "World",0
    Dim wAddress As Word

'
' Table of address's
'
    Dim AddrTable As Code = As Word MyString1, MyString2

    RPOR7 = 3                         ' Make PPS Pin RP14 U1TX
    wAddress = CRead16 AddrTable[0]   ' Locate the address of first string
    While                             ' Create an infinite loop
        MyByte = cPtr8(wAddress++)    ' Read each character from code memory
        If MyByte = 0 Then Break      ' Exit when null found
        HRSOut MyByte                 ' Display the character
    Wend                              ' Close the loop
    HRSOut 13
    wAddress = CRead16 AddrTable[1]   ' Locate the address of second string
    While                             ' Create an infinite loop
        MyByte = cPtr8(wAddress++)    ' Read each character from code memory
        If MyByte = 0 Then Break      ' Exit when null found
        HRSOut MyByte                 ' Display the character
    Wend                              ' Close the loop
```

Note.

It is not recommended to use **Cdata** in a new program, and may be dropped from future compiler versions. It is recommended to use the **Dim As Code** construct.

See also : **cPtr8, cPtr16, cPtr32, cPtr64, Cread8, Cread16, Cread32, Cread64, Dim.**

Circle

Syntax
Circle *Set_Clear*, *Xpos*, *Ypos*, *Radius*

Overview
Draw a circle on a graphic LCD.

Operands
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. If using a colour graphic LCD, this parameter holds the 16-bit colour of the pixel.
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 65535.

KS0108 LCD example
```
' Draw circle at pos 63,32 with radius of 20 pixels on a Samsung KS0108 LCD

  Device = 24FJ64GA002
  Declare Xtal = 16

  Declare LCD_Type = Samsung       ' Setup for a Samsung KS0108 graphic LCD
  Declare LCD_DTPort = PORTB.Byte0 ' Use the first 8-bits of PORTB
  Declare LCD_CS1Pin = PORTB.8
  Declare LCD_CS2Pin = PORTB.9
  Declare LCD_ENPin = PORTB.10
  Declare LCD_RSPin = PORTB.11
  Declare LCD_RWPin = PORTB.12

  Dim Xpos as Byte
  Dim Ypos as Byte
  Dim Radius as Byte
  Dim SetClr as Byte

  DelayMs 100               ' Wait for things to stabilise
  Cls                       ' Clear the LCD
  Xpos = 63
  Ypos = 32
  Radius = 20
  SetClr = 1
  Circle SetClr, Xpos, Ypos, Radius
```

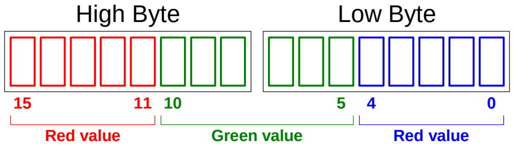

ILI9320 colour graphic LCD example

```
' Demonstrate the circle command with a colour LCD
'
  Device = 24EP128MC202
  Declare Xtal = 140.03
'
' Setup the Pins used by the ILI9320 graphic LCD
'
  Declare LCD_DTPort = PORTB.Byte0    ' Use the first 8-bits of PORTB
  Declare LCD_CSPin = PORTB.8         ' Connect to the LCD's CS pin
  Declare LCD_RDPin = PORTB.9         ' Connect to the LCD's RD pin
  Declare LCD_RSPin = PORTB.10        ' Connect to the LCD's RS pin
  Declare LCD_WRPin = PORTA.3         ' Connect to the LCD's WR pin

  Include "ILI9320.inc"      ' Load the ILI9320 routines into the program

    Dim wRadius As Word      ' Create a variable for the circle's radius

'--------------------------------------------------------------------------
Main:
' Configure the internal Oscillator to operate the device at 140.03MHz
'
  PLL_Setup(76, 2, 2, $0300)

  Cls clYellow                  ' Clear the LCD with the colour yellow
  For wRadius = 0 To 319
    Circle clBrightCyan, 120, 160, wRadius ' Draw a series of circles
  Next
'--------------------------------------------------------------------------
' Configure for internal 7.37MHz oscillator with PLL
' OSC pins are general purpose I/O
'
  Config FGS = GWRP_OFF, GCP_OFF
  Config FOSCSEL = FNOSC_FRCPLL, IESO_ON, PWMLOCK_OFF
  Config FOSC = POSCMD_NONE, OSCIOFNC_ON, IOL1WAY_OFF, FCKSM_CSDCMD
  Config FWDT = WDTPOST_PS256, WINDIS_OFF, PLLKEN_ON, FWDTEN_OFF
  Config FPOR = ALTI2C1_ON, ALTI2C2_OFF
  Config FICD = ICS_PGD1, JTAGEN_OFF
```

Notes.

Because of the aspect ratio of the pixels on the KS0108 graphic LCD (approx 1.5 times higher than wide) the circle will appear elongated.

With an ILI9320 320x240 pixel colour graphic LCD, the colour is a 16-bit value formatted in RGB565, where the upper 5-bits represent the red content, the middle 6-bits represent the green content, and the lower 5-bits represent the blue content. As illustrated below:



For convenience, there are several colours defined within the ILI9320.inc file. These are:

```
clBlack
clBrightBlue
clBrightGreen
clBrightCyan
clBrightRed
clBrightMagenta
clBrightYellow
clBlue
clGreen
clCyan
clRed
clMagenta
clBrown
clLightGray
clDarkGray
clLightBlue
clLightGreen
clLightCyan
clLightRed
clLightMagenta
clYellow
clWhite
```

More constant values for colours can be added by the user if required.

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.

Operands
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 SR.0         ' Clear the carry flag
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.

Operands
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 = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600      ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14   ' Select the pin used for TX with USART1

  Dim MyByte as Byte
  Dim Index as Byte

  RPOR7 = 3                       ' Make PPS Pin RP14 U1TX
  MyByte = %11111111
  While                           ' Create an infinite loop
    For Index = 0 to 7            ' Create a loop for 8 bits
      ClearBit MyByte,Index       ' Clear each bit of MyByte
      Hrsout Bin8 MyByte  , 13    ' 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 MyByte,Index         ' Set each bit of MyByte
      Hrsout Bin8 MyByte, 13      ' 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. 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

or if using a colour graphic LCD

Cls *Colour*

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 a Samsung KS0108 graphic LCD
  Device = 24FJ64GA002
  Declare Xtal = 16

  Declare LCD_Type = Samsung       ' Setup for a Samsung KS0108 graphic LCD
  Declare LCD_DTPort = PORTB.Byte0 ' Use the first 8-bits of PORTB
  Declare LCD_CS1Pin = PORTB.8
  Declare LCD_CS2Pin = PORTB.9
  Declare LCD_ENPin = PORTB.10
  Declare LCD_RSPin = PORTB.11
  Declare LCD_RWPin = PORTB.12

  DelayMs 100                ' Wait for things to stabilise
  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
```

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.
  Device = 24FJ64GA002
  Declare Xtal = 16

  Include "T6963C.inc"     ' Load the T6963C routines into the program

  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
```

See also : **Cursor, Print, Toshiba_Command.**

Config

Syntax
Config *Register Name = Fuse Name, Fuse Name {,Fuse Name}*

Overview
Enable or Disable particular fuse settings for the device being used.

Operands
Register Name is the designated name of the fuse register within the microcontroller. These vary from device family to device family.
Fuse Name is a list of comma delimited texts that represent the fuse to enable or disable accordingly.

At the time of writing, the standard PIC24F® devices use the texts **Config1**, **Config2**, **Config3**, and **Config4**, in order to designate a fuse register, depending on the type. However, some of the newer PIC24FV® devices and all the PIC24H® and PIC24E® and dsPIC33® devices use the texts **FBS**, **FGS**, **FOSCSEL**, **FOSC**, **FWDT**, **FPOR**, **FICD**, **FDS** for fuse register designations.

Example 1
```
' Alter the fuses for a standard PIC24F device (24FJ64GA002)
  Config Config1 = JTAGEN_OFF, GCP_OFF, GWRP_OFF, BKBUG_OFF, COE_OFF,_
                   ICS_PGx1, FWDTEN_OFF, WINDIS_OFF, FWPSA_PR128,_
                   WDTPOST_PS256
  Config Config2 = IOL1WAY_OFF, COE_OFF, IESO_OFF, FNOSC_PRI,_
                   FCKSM_CSDCMD, OSCIOFNC_OFF, POSCMOD_HS
```

Example 2
```
' Alter the fuses for a PIC24H device (24HJ128GP502)
  Config FBS = BWRP_WRPROTECT_OFF, BSS_NO_FLASH, BSS_NO_BOOT_CODE
  Config FSS = SWRP_WRPROTECT_OFF, SSS_NO_FLASH, RSS_NO_SEC_RAM
  Config FGS = GWRP_OFF, GCP_OFF
  Config FOSCSEL = FNOSC_FRCPLL, IESO_OFF
  Config FOSC = POSCMD_HS, OSCIOFNC_OFF, IOL1WAY_OFF, FCKSM_CSDCMD
  Config FWDT = WDTPOST_PS256, WINDIS_OFF, FWDTEN_OFF
  Config FPOR = FPWRT_PWR128, ALTI2C_OFF
  Config FICD = ICS_PGD1, JTAGEN_OFF
```

Notes.
The device's PPI file has the required fuse designators in its [FUSESTART] section and a list of the valid fuse names in its [CONFIGSTART] section. The default location of the PPI files is:

For Windows XP and Windows 7 (32-bit)
C:\Program Files\ProtonIDE\PDS\Includes\PPI

For Windows 7 (64-bit)
C:\Program Files (x86)\ProtonIDE\PDS\Includes\PPI

For detailed information concerning the configuration fuses, refer to the microcontroller's datasheet.

The compiler's default fuse settings are for an external oscillator with no PLL. In order for the **Sleep** command's timing to remian correct, always use the fuse setting `WDTPOST_PS256`

Below are three examples of using the Config directive:

Example 1

```
' PIC24F external 8MHz crystal operating at 32MHz using PLL
'
  Device = 24FJ64GA002
  Declare Xtal = 32

  CLKDIV = 0                    ' CPU peripheral clock ratio set to 1:1
  OSCCON.Byte1 = %00010000      ' Enable 4 x PLL '
'
' Flash an LED connected to PORTA.0
'
  While
    High PORTA.0
    DelayMS 500
    Low PORTA.0
    DelayMS 500
  Wend
'
' Configure for external oscillator with PLL
'
  Config Config1 = JTAGEN_OFF, GCP_OFF, BKBUG_OFF,_
                   COE_OFF, ICS_PGx1, FWDTEN_OFF, WINDIS_OFF,_
                   FWPSA_PR128, WDTPOST_PS256
  Config Config2 = IOL1WAY_OFF, IESO_OFF, FNOSC_PRIPLL,_
                   FCKSM_CSECME,OSCIOFNC_OFF, POSCMOD_HS
```

Example 2

```
' PIC24F internal 8MHz oscillator operating at 32MHz using PLL
'
  Device = 24FJ64GA002
  Declare Xtal = 32

  CLKDIV = 0                    ' CPU peripheral clock ratio set to 1:1
  OSCCON.Byte1 = %00010000      ' Enable 4 x PLL
'
' Flash an LED connected to PORTA.0
'
  While
    High PORTA.0
    DelayMS 500
    Low PORTA.0
    DelayMS 500
  Wend
'
' Configure for internal 8MHz oscillator with PLL
' OSC pins operate as general purpose I/O
'
  Config Config1 = JTAGEN_OFF, GCP_OFF, BKBUG_OFF,_
                   COE_OFF, ICS_PGx1, FWDTEN_OFF, WINDIS_OFF,_
                   FWPSA_PR128, WDTPOST_PS256
  Config Config2 = IOL1WAY_OFF, IESO_OFF, FNOSC_PRIPLL,_
                   FCKSM_CSDCMD, OSCIOFNC_OFF, POSCMOD_NONE
```

Example 3

```
' PIC24H internal 7.37MHz oscillator operating at 79.23MHz using PLL
'
    Device = 24HJ128GP502
    Declare Xtal = 79.23
'-------------------------------------------------------------------
Main:
' Configure the Oscillator to operate the device at 79.23MHz
' Fosc = (7.37 * 43) / (2 * 2) = 79.23MHz (40 MIPS)
'
  PLL_Setup(43, 2, 2, $0300)
'
' Flash an LED connected to PORTA.0
'
  While
    High PORTA.0
    DelayMS 500
    Low PORTA.0
    DelayMS 500
  Wend
'
' Configure for internal 7.37MHz oscillator with PLL
' OSC pins operate as general purpose I/O
'
    Config FBS = BWRP_WRPROTECT_OFF
    Config FSS = SWRP_WRPROTECT_OFF
    Config FGS = GWRP_OFF
    Config FOSCSEL = FNOSC_FRCPLL, IESO_OFF
    Config FOSC = POSCMD_NONE, OSCIOFNC_OFF, IOL1WAY_OFF, FCKSM_CSECME
    Config FWDT = WDTPOST_PS256, WINDIS_OFF, FWDTEN_OFF
    Config FPOR = FPWRT_PWR128, ALTI2C_OFF
    Config FICD = ICS_PGD1, JTAGEN_OFF
```

Example 4

```
' PIC24E internal 7.37MHz oscillator operating at 140.03MHz using PLL
'
    Device = 24EP128MC202
    Declare Xtal = 140.03
'-------------------------------------------------------------------
Main:
' Configure the Oscillator to operate the device at 140.03MHz (70 MIPS)
' Fosc = (7.37 * 76) / (2 * 2) = 140.03MHz
'
  PLL_Setup(76, 2, 2, $0300)
'
' Flash an LED connected to PORTA.0
'
  While
    High PORTA.0
    DelayMS 500
    Low PORTA.0
    DelayMS 500
  Wend
'
' Configure for internal 7.37MHz oscillator with PLL
' OSC pins operate as general purpose I/O
'
  Config FGS = GWRP_OFF
  Config FOSCSEL = FNOSC_FRCPLL, IESO_OFF, PWMLOCK_OFF
  Config FOSC = POSCMD_NONE, OSCIOFNC_ON, IOL1WAY_OFF, FCKSM_CSECME
  Config FWDT = WDTPOST_PS256, WINDIS_OFF, PLLKEN_ON, FWDTEN_OFF
  Config FPOR = ALTI2C1_ON, ALTI2C2_OFF
  Config FICD = ICS_PGD1, JTAGEN_OFF
```


Example 5

```
' PIC33F internal 7.37MHz oscillator operating at 79.23MHz using PLL
'
    Device = 33FJ128MC802
    Declare Xtal = 79.23
'--------------------------------------------------------------------
Main:
' Configure the Oscillator to operate the device at 79.23MHz
' Fosc = (7.37 * 43) / (2 * 2) = 79.23MHz (40 MIPS)
'
  PLL_Setup(43, 2, 2, $0300)
'
' Flash an LED connected to PORTA.0
'
  While
    High PORTA.0
    DelayMS 500
    Low PORTA.0
    DelayMS 500
  Wend
'
' Configure for internal 7.37MHz oscillator with PLL
' OSC pins operate as general purpose I/O
'
    Config FBS = BWRP_WRPROTECT_OFF
    Config FGS = GWRP_OFF
    Config FOSCSEL = FNOSC_FRCPLL, IESO_OFF
    Config FOSC = POSCMD_NONE, OSCIOFNC_OFF, IOL1WAY_OFF, FCKSM_CSECME
    Config FWDT = WDTPOST_PS256, WINDIS_OFF, FWDTEN_OFF
    Config FPOR = FPWRT_PWR128, ALTI2C_OFF
    Config FICD = ICS_PGD1, JTAGEN_OFF
```

Note.

The **PLL_Setup** helper macro can be found within the device's ".def" file. It is required because the dsPIC33E®, PIC24E® and PIC24H® devices need an unlock sequence before writing to the OSCCON SFR, unlike the PIC24F® devices, that can write directly to the OSCCON SFR.

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 = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600      ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14   ' Select the pin for TX with USART1

  Dim Index as Byte

  RPOR7 = 3                       ' Make PPS Pin RP14 U1TX

  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.**

Counter

Syntax
Variable **= Counter** *Pin*, *Period*

Overview
Count the number of pulses that appear on *pin* during *period*, and store the result in *variable*.

Operands
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.

  Device = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600     ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14  ' Select the pin for TX with USART1

  Dim MyWord as Word            ' Declare a word size variable
  Symbol Pin = PORTA.0          ' Assign the input pin to PORTA.0

  RPOR7 = 3                    ' Make PPS Pin RP14 U1TX
  While                         ' Create an infinite loop
    MyWord = Counter Pin, 100 ' Variable MyWord now contains the Count
    Hrsout Dec MyWord, 13     ' Display the decimal result
  Wend                          ' Do it forever
```

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.

See also : Pulseln, Rcin.

cPtr8, cPtr16, cPtr32, cPtr64

Syntax
Variable **= cPtr8** *(Address)*
Variable **= cPtr16** *(Address)*
Variable **= cPtr32** *(Address)*
Variable **= cPtr64** *(Address)*

Overview
Indirectly read code memory using a variable to hold the 16-bit or 32-bit address.

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 retreiving it's RAM location.
- (MyAddress --) Post decrement MyAddress after retreiving it's RAM location.
- (++MyAddress) Pre increment MyAddress before retreiving it's RAM location.
- (--MyAddress) Pre decrement MyAddress before retreiving 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.
cPtr64 will retrieve a value with an optional 64-bit post or pre increment or decrement.

8-bit Example.
```
'
' Read 8-bit values indirectly from code memory
'
  Device = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600          ' UART1 baud rate
  Declare Hrsout1_Pin = PORTB.14       ' Select pin to be used for TX
'
' 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:
  RPOR7 = 3                          ' Make PPS Pin RP14 U1TX
'
' 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 = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600        ' UART1 baud rate
  Declare Hrsout1_Pin = PORTB.14     ' Select pin is to be used for TX
'
' 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:
  RPOR7 = 3                         ' Make PPS Pin RP14 U1TX
'
' 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 = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600        ' UART1 baud rate
  Declare Hrsout1_Pin = PORTB.14     ' Select pin is to be used for TX
'
' 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:
  RPOR7 = 3                         ' Make PPS Pin RP14 U1TX
'
' 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, Cread64, Ptr8, Ptr16, Ptr32, Ptr64.**

Cread8, Cread16, Cread32, Cread64

Syntax
Variable **= Cread8** *Label* **[** *Offset Variable* **]**
or
Variable **= Cread16** *Label* **[** *Offset Variable* **]**
or
Variable **= Cread32** *Label* **[** *Offset Variable* **]**
or
Variable **= Cread64** *Label* **[** *Offset Variable* **]**

Overview
Read an 8, 16, 32 or 64-bit value from a code memory table using an offset of *Offset Variable* and place into *Variable*.

Cread8 will access 8-bit values from a code memory table.
Cread16 will access 16-bit values from a code memory table.
Cread32 will access 32-bit values from a code memory table, this also includes 32-bit floating point values.
Cread64 will access 64-bit values from a code memory table.

Operands
Variable is a user defined variable.
Label is a label name given to the code memory table 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 code memory table.

Cread8 Example
```
' Extract the second value from within an 8-bit code memory table
'
  Device = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600      ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14   ' Select the pin for TX with USART1

  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
'
' Create a table containing only 8-bit values
'
  Dim Byte_Table as Code = as Byte 100, 200

  RPOR7 = 3                  ' Make PPS Pin RP14 U1TX

  Offset = 1                 ' Point to second value in the code memory table
'
' Read the 8-bit value pointed to by Offset
'
  Result = Cread8 Byte_Table[Offset]
  Hrsout Dec Result, 13      ' Display the decimal result
```

Cread16 Example

```
' Extract the second value from within a 16-bit code memory table
'
  Device = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600     ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14  ' Select the pin for TX with USART1

  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
'
' Create a table containing only 16-bit values
'
  Dim WordTable as Code = as Word 1234, 5678

  RPOR7 = 3                ' Make PPS Pin RP14 U1TX
  Offset = 1               ' Point to the second value in the code table
'
' Read the 16-bit value pointed to by Offset
'
  Result = Cread16 WordTable[Offset]
  Hrsout Dec Result, 13    ' Display the decimal result
```

Cread32 Example

```
' Extract the second value from within a 32-bit code memory table
'
  Device = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600     ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14  ' Select the pin for TX with USART1

  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
'
' Create a table containing only 32-bit values
'
  Dim DwordTable as Code = as Dword 12340, 56780

  RPOR7 = 3               ' Make PPS Pin RP14 U1TX
  Offset = 1              ' Point to the second value in the code table
'
' Read the 32-bit value pointed to by Offset
'
  Result = Cread32 DwordTable[Offset]
  Hrsout Dec Result, 13 ' Display the decimal result
```

See also : **Dim as code, cPtr8, cPtr16, cPtr32, cPtr64.**

Cursor

Syntax
Cursor *Line, Position*

Overview
Move the cursor position on an Alphanumeric or Graphic LCD to a specified line (ypos) and position (xpos).

Operands
Line is a constant, variable, or expression that corresponds to the line (Ypos) number from 1 to maximum lines (0 to maximum Y resolution 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
```
Device = 24FJ64GA002
Declare Xtal = 16

Declare LCD_DTPin = PORTB.4
Declare LCD_RSPin = PORTA.0
Declare LCD_ENPin = PORTA.1
Declare LCD_Lines = 4
Declare LCD_Interface = 4

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

```
Device = 24FJ64GA002
Declare Xtal = 16

Declare LCD_DTPin = PORTB.4
Declare LCD_RSPin = PORTA.0
Declare LCD_ENPin = PORTA.1
Declare LCD_Lines = 4
Declare LCD_Interface = 4

Dim Xpos as Byte
Dim Ypos as Byte

While                       ' Create an infinite loop
  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
Wend                        ' Do it forever
```

Example 2 displays an asterisk character moving around the perimeter of a 2-line by 16 character LCD.

See also : Cls, Print

Dec

Syntax
Dec *Variable*

Overview
Decrement a variable i.e. Var1 = Var1 - 1

Operands
Variable is a user defined variable

Example
```
Device = 24FJ64GA002
Declare Xtal = 16
Declare Hserial_Baud = 9600     ' USART1 baud rate
Declare Hrsout1_Pin = PORTB.14  ' Select the pin for TX with USART1

Dim MyWord as Word

RPOR7 = 3                      ' Make PPS Pin RP14 U1TX
MyWord = 11
Repeat
  Dec MyWord
  Hrsout Dec MyWord, 13
  DelayMs 200
Until MyWord = 0
```

The above example shows the equivalent to the **For**-**Next** loop: -

```
For MyWord = 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.

Operands
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.

Note.
The **Declare** directive is mandatory and must precede the texts, otherwise a syntax error will be produced.

The **Declare** directive usually alters the corresponding library subroutine at compile time. 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.

Misc Declares.

Declare WatchDog = On or Off, or True or False, or 1, 0
The **WatchDog Declare** directive enables or disables the **ClrWdt** mnemonic within strategic locations of the compiler's library subroutines. Unlike Proton for 8-bit microcontrollers, it does **not** enable the watchdog fuse. This must be done by using the **Config** directive. The default for the compiler is **WatchDog Off**, therefore, if the watchdog timer is required, then this **Declare** will need to be invoked.

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 reminder messages at key points in the code as and when required.

Declare Access_Upper_64K = On or Off, or True or False, or 1, 0
Some PIC24® and dsPIC® devices have very large amounts of code memory storage, however, because the architecture of the devices is 16-bit, the largest address that can be accessed with a single mnemonic is 65535 bytes. When this address is exceeded, the device's TBLPAG SFR must be loaded with the 17th, 18th, up to 24th bit of the address.

Note that this only applies to data stored in code memory using the **Cdata** directive or very large data segments using the **Dim as Code** directive. It does not usually affect normal commands or mnemonics.

When the **Access_Upper_64K** declare is used, the compiler will add code that manipulates the TBLPAG SFR, however, this will impact on the code size produced by the compiler.

Adin Declares.

Declare Adin_Tad c1_FOSC, c2_FOSC, c4_FOSC, c8_FOSC, c16_FOSC, c32_FOSC, c64_FOSC, or cFRC.
Sets the ADC's clock source.

All compatible devices have multiple options for the clock source used by the ADC peripheral. 1_FOSC, 2_FOSC, 4_FOSC, 8_FOSC, 16_FOSC, 32_FOSC, and 64_FOSC are ratios of the external oscillator, while FRC is the device's internal RC oscillator.

Care must be used when issuing this **Declare**, as the wrong type of clock source may result in poor accuracy, or no conversion at all. If in doubt use FRC which will produce a slight reduction in accuracy 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 2 to 100. This allows adequate charge time without loosing 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 microcontroller. 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 microcontroller. 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 I^2C 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 I^2C 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 Declares.

Declare HSDA_Pin Port **.** Pin
Declares the port and pin used for the data line (SDA). The location of the port and pin used for hardware I^2C can be altered by the fuse configurations. If the declare is not used in the program, it will default to the standard pin configuration.

Declare HSCL_Pin Port **.** Pin
Declares the port and pin used for the clock line (SCL). The location of the port and pin used for hardware I^2C can be altered by the fuse configurations. If the declare is not used in the program, it will default to the standard pin configuration.

Declare Hbus_Bitrate Constant 100, 400, 1000 etc.
The standard speed for the I^2C bus is 100KHz. Some devices use a higher bus speed of 400KHz. The above **Declare** allows the I^2C 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.

USART1 Declares for Hserin, Hserout, Hrsin and Hrsout.

Declare HRsout_Pin Port **.** Pin
Declares the port and pin used for USART1 transmission (TX). The location of the port and pin is dictated by the device's PPS (Peripheral Pin Select) options. Note that this declare will not alter any PPS (Peripheral Pin Select) SFRs.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare HRsin_Pin Port **.** Pin
Declares the port and pin used for USART1 reception (RX). The location of the port and pin is dictated by the device's PPS (Peripheral Pin Select) options. Note that this declare will not alter any PPS (Peripheral Pin Select) SFRs.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare Hserial_Baud Constant value
Sets the Baud rate that will be used to transmit or receive a byte 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 9600 baud.

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 (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.

Declare Hrsout_Pace 0 to 65535 microseconds (us)
Implements a delay between characters transmitted by the **Hrsout** or **HSerout** 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 **Hrsout** or **HSerout**.

If the **Declare** is not used in the program, then the default is no delay between characters.

USART2 Declares for use with Hrsin2, Hserin2, Hrsout2 and Hserout2.

Declare HRsout2_Pin Port **.** Pin
Declares the port and pin used for USART2 transmission (TX). The location of the port and pin is dictated by the device's PPS (Peripheral Pin Select) options. Note that this declare will not alter any PPS (Peripheral Pin Select) SFRs.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare HRsin2_Pin Port **.** Pin
Declares the port and pin used for USART2 reception (RX). The location of the port and pin is dictated by the device's PPS (Peripheral Pin Select) options. Note that this declare will not alter any PPS (Peripheral Pin Select) SFRs.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare Hserial2_Baud Constant value
Sets the Baud rate that will be used to transmit or receive a byte 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 9600 baud.

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 (7data 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.

```
Declare Hserial2_Clear = On
```

Declare Hrsout2_Pace 0 to 65535 microseconds (us)
Implements a delay between characters transmitted by the **Hrsout2** or **HSerout2** 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 **Hrsout2** or **HSerout2**.

If the **Declare** is not used in the program, then the default is no delay between characters.

USART3 Declares for use with Hrsin3, Hserin3, Hrsout3 and Hserout3.

Declare HRsout3_Pin Port **.** Pin
Declares the port and pin used for USART3 transmission (TX). The location of the port and pin is dictated by the device's PPS (Peripheral Pin Select) options. Note that this declare will not alter any PPS (Peripheral Pin Select) SFRs.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare HRsin3_Pin Port **.** Pin
Declares the port and pin used for USART3 reception (RX). The location of the port and pin is dictated by the device's PPS (Peripheral Pin Select) options. Note that this declare will not alter any PPS (Peripheral Pin Select) SFRs.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare Hserial3_Baud Constant value
Sets the Baud rate that will be used to transmit or receive a byte 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 9600 baud.

Declare Hserial3_Parity Odd or Even
Enables/Disables parity on the serial port. For **Hrsout3**, **Hrsin3**, **Hserout3** and **Hserin3**. 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 **Hserial3_Parity** declare.

```
Declare Hserial3_Parity = Even   ' Use if even parity desired
Declare Hserial3_Parity = Odd    ' Use if odd parity desired
```

Declare Hserial3_Clear On or Off
Clear the overflow error bit before commencing a read.

```
Declare Hserial3_Clear = On
```

Declare Hrsout3_Pace 0 to 65535 microseconds (us)
Implements a delay between characters transmitted by the **Hrsout3** or **HSerout3** 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 **Hrsout3** or **HSerout3**.

If the **Declare** is not used in the program, then the default is no delay between characters.

USART4 Declares for use with Hrsin4, Hserin4, Hrsout4 and Hserout4.

Declare HRsout4_Pin Port **.** Pin
Declares the port and pin used for USART4 transmission (TX). The location of the port and pin is dictated by the device's PPS (Peripheral Pin Select) options. Note that this declare will not alter any PPS (Peripheral Pin Select) SFRs.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare HRsin4_Pin Port **.** Pin
Declares the port and pin used for USART4 reception (RX). The location of the port and pin is dictated by the device's PPS (Peripheral Pin Select) options. Note that this declare will not alter any PPS (Peripheral Pin Select) SFRs.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare Hserial4_Baud Constant value
Sets the Baud rate that will be used to transmit or receive a byte 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 9600 baud.

Declare Hserial4_Parity Odd or Even
Enables/Disables parity on the serial port. For **Hrsout4**, **Hrsin4**, **Hserout4** and **Hserin4**. 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 **Hserial4_Parity** declare.

```
Declare Hserial4_Parity = Even   ' Use if even parity desired
Declare Hserial4_Parity = Odd    ' Use if odd parity desired
```

Declare Hserial3_Clear On or Off
Clear the overflow error bit before commencing a read.

```
Declare Hserial3_Clear = On
```

Declare Hrsout4_Pace 0 to 65535 microseconds (us)
Implements a delay between characters transmitted by the **Hrsout4** or **HSerout4** 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 **Hrsout4** or **HSerout4**.

If the **Declare** is not used in the program, then the default is no delay between characters.

Hpwm Declares.

Some devices 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)
Declare CCP3_Pin Port.Pin ' Select Hpwm port and bit for CCP3 module (ch 3)
Declare CCP4_Pin Port.Pin ' Select Hpwm port and bit for CCP4 module (ch 4)
Declare CCP5_Pin Port.Pin ' Select Hpwm port and bit for CCP5 module (ch 5)
```


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 microcontroller 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  ' Use a 4-line interface on low byte of PORTB
Declare LCD_DTPin PORTB.0  ' Use an 8-line interface on low byte of PORTB
Declare LCD_DTPin PORTB.12' Use a 4-line interface on high byte of PORTB
Declare LCD_DTPin PORTB.8  ' Use an 8-line interface on high byte of PORTB
```

In the above examples, PORTB is only a personal preference. The LCD's DT lines can be attached to any valid port on the microcontroller.

There is no default setting for this **Declare** and it must be used within the BASIC program.

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.

There is no default setting for this **Declare** and it must be used within the BASIC program.

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.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_Interface 4 or 8
Inform the compiler as to whether a 4-line or 8-line interface is required by the LCD.

There is no default setting for this **Declare** and it must be used within the BASIC program.

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.

There is no default setting for this **Declare** and it must be used within the BASIC program.

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 65535
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 Alpha or **Graphic** or **Samsung** or **Toshiba** or **Colour**
Inform the compiler as to the type of LCD that the **Print** command will output to. If **Graphic**, or **Samsung** is chosen then any output by the **Print** command will be directed to a graphic LCD based on the Samsung KS0108 chipset. The text **Toshiba**, will direct the output to a graphic LCD based on the Toshiba T6963 chipset. The text Colour will direct the output to an ILI9320 Colour Graphic LCD. The text **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** etc.

KS0108 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_RWPin Port **.** Pin
Assigns the Port and Pin that the graphic LCD's RW line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_ENPin Port **.** Pin
Assigns the Port and Pin that the graphic LCD's EN line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_RSPin Port **.** Pin
Assigns the Port and Pins that the graphic LCD's RS line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_CS1Pin Port **.** Pin
Assigns the Port and Pin that the graphic LCD's CS1 line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_CS2Pin Port **.** Pin
Assigns the Port and Pin that the graphic LCD's CS2 line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

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 16383 cycles.
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.

If the **Declare** is not used in the program, then the cycles delay is determined by the oscillator used.

Toshiba T6963C 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 subroutines.

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. There is no default setting for this **Declare.**

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. There is no default setting for this **Declare**.

ILI9320 Colour 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_CSPin Port **.** Pin
Assigns the Port and Pin that the graphic LCD's CS line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_RSPin Port **.** Pin
Assigns the Port and Pins that the graphic LCD's RS 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.

ADS7846 Touch Screen controller Declares.

Declare Touch_CSPin Port **.** Pin
Assigns the Port and Pin that will attach to the ADS7846 chip's CS pin.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare Touch_CLKPin Port **.** Pin
Assigns the Port and Pin that will attach to the ADS7846 chip's CLK pin.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare Touch_DINPin Port **.** Pin
Assigns the Port and Pin that will attach to the ADS7846 chip's DIN pin.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare Touch_DOUTPin Port **.** Pin
Assigns the Port and Pin that will attach to the ADS7846 chip's DOUT pin.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Keypad Declare.

Declare Keypad_Port Port
Assigns the Port that the keypad is attached to.

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 microcontroller.

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 microcontroller.

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 etc...

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 and above.

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 frequency. The active state is held for a minimum of 2 microseconds, again depending on the oscillator. 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.

Stack Declares.

Declare Stack_Size = 20 to n (in words)
The compiler sets the default size of the microcontroller's stack to 60 words (120 bytes). This can be increased or decreased as required, as long as it fits within the RAM available. The compiler places a minimum limit of 20 for stack size. If the stack overflows or underflows, the microcontroller will trigger an exception. The compiler's command library routines make extensive use of the stack for saving and restoring WREG SFRs, therefore, make sure the stack is large enough to accommodate all the **Gosub**/**Return** commands, as well as temporary data used.

When 64-bit floating point **Double** variables are being used in trigonometry routines, it is important to increase the stack size because the library routines use the stack intensively as temporary storage. A stack size of 200 words will usually suffice. If the program resets intermittently, the stack size is too small and the microcontroller is executing an over/under stack exception.

Declare Stack_Expand = 1 or 0 or On or Off
Whenever an interrupt handler is used within a BASIC program, it must context save and restore critical SFRs and variables that would otherwise get overwritten. It uses the microcontroller's stack for temporary storage of the SFRs and variables, therefore the stack will increase with every interrupt handler used within the program. If this behaviour is undesirable, the above declare will disable it. However, the user must make sure that the stack is large enough to accommodate the storage, otherwise an exception will be triggered by the microcontroller.

Oscillator Frequency Declare.

Declare Xtal = Frequency (in MHz).

Inform the compiler what frequency oscillator is being used. For example:

```
  Declare Xtal = 7.37
```

or

```
  Declare Xtal = 80
```

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.

Note

The **Xtal** declare will not alter any fuse settings or SFRs (Special Function Registers) relating to the oscillator setup. There is no default value if the **Declare** is not issued in a program, and it should be considered as a mandatory addition to the code.

PIC24® and dsPIC33® devices have a multitude of oscillator options, therefore, they cannot all be detailed in this manual. However, shown below are some examples that illustrate methods of using an external crystal and the internal oscillator, both with and without the PLL multiplier.

Example 1

```
' PIC24F external 8MHz crystal operating at 32MHz using PLL
'
  Device = 24FJ64GA002
  Declare Xtal = 32

  CLKDIV = 0                    ' CPU peripheral clock ratio set to 1:1
  OSCCON.Byte1 = %00010000      ' Enable 4 x PLL '
'
' Flash an LED connected to PORTA.0
'
  While
    High PORTA.0
    DelayMS 500
    Low PORTA.0
    DelayMS 500
  Wend
'
' For external oscillator with PLL
'
  Config Config1 = JTAGEN_OFF, GCP_OFF, GWRP_OFF, BKBUG_OFF,_
                   COE_OFF, ICS_PGx1, FWDTEN_OFF, WINDIS_OFF,_
                   FWPSA_PR128, WDTPOST_PS256
  Config Config2 = IOL1WAY_OFF, IESO_OFF, FNOSC_PRIPLL,_
                   FCKSM_CSDCMD,OSCIOFNC_OFF, POSCMOD_HS
```


Example 2

```
' PIC24F internal 8MHz oscillator operating at 32MHz using PLL
'
  Device = 24FJ64GA002
  Declare Xtal = 32

  CLKDIV = 0                    ' CPU peripheral clock ratio set to 1:1
  OSCCON.Byte1 = %00010000      ' Enable 4 x PLL
'
' Flash an LED connected to PORTA.0
'
  While
    High PORTA.0
    DelayMS 500
    Low PORTA.0
    DelayMS 500
  Wend
'
' For internal 8MHz oscillator with PLL
' OSC pins operate as general purpose I/O
'
  Config Config1 = JTAGEN_OFF, GCP_OFF, BKBUG_OFF,_
                   COE_OFF, ICS_PGx1, FWDTEN_OFF, WINDIS_OFF,_
                   FWPSA_PR128, WDTPOST_PS256
  Config Config2 = IOL1WAY_OFF, IESO_OFF, FNOSC_PRIPLL,_
                   FCKSM_CSECME, OSCIOFNC_OFF, POSCMOD_NONE
```

Example 3

```
' PIC24H internal 7.37MHz oscillator operating at 79.23MHz using PLL
'
    Device = 24HJ128GP502
    Declare Xtal = 79.23
'-------------------------------------------------------------------
Main:
' Configure the Oscillator to operate the device at 79.23MHz
' Fosc = (7.37 * 43) / (2 * 2) = 79.23MHz (40 MIPS)
'
  PLL_Setup(43, 2, 2, $0300)
'
' Flash an LED connected to PORTA.0
'
  While
    High PORTA.0
    DelayMS 500
    Low PORTA.0
    DelayMS 500
  Wend
'
' For internal 7.37MHz oscillator with PLL
' OSC pins operate as general purpose I/O
'
    Config FBS = BWRP_WRPROTECT_OFF
    Config FSS = SWRP_WRPROTECT_OFF
    Config FGS = GWRP_OFF
    Config FOSCSEL = FNOSC_FRCPLL, IESO_OFF
    Config FOSC = POSCMD_NONE, OSCIOFNC_OFF, IOL1WAY_OFF, FCKSM_CSECME
    Config FWDT = WDTPOST_PS256, WINDIS_OFF, FWDTEN_OFF
    Config FPOR = FPWRT_PWR128, ALTI2C_OFF
    Config FICD = ICS_PGD1, JTAGEN_OFF
```

Example 4

```
' PIC24E internal 7.37MHz oscillator operating at 140.03MHz using PLL
'
    Device = 24EP128MC202
    Declare Xtal = 140.03
'-------------------------------------------------------------------
Main:
' Configure the Oscillator to operate the device at 140.03MHz (70 MIPS)
' Fosc = (7.37 * 76) / (2 * 2) = 140.03MHz
'
  PLL_Setup(76, 2, 2, $0300)
'
' Flash an LED connected to PORTA.0
'
  While
    High PORTA.0
    DelayMS 500
    Low PORTA.0
    DelayMS 500
  Wend
'
' For internal 7.37MHz oscillator with PLL
' OSC pins operate as general purpose I/O
'
  Config FGS = GWRP_OFF
  Config FOSCSEL = FNOSC_FRCPLL, IESO_OFF, PWMLOCK_OFF
  Config FOSC = POSCMD_NONE, OSCIOFNC_ON, IOL1WAY_OFF, FCKSM_CSECME
  Config FWDT = WDTPOST_PS256, WINDIS_OFF, PLLKEN_ON, FWDTEN_OFF
  Config FPOR = ALTI2C1_ON, ALTI2C2_OFF
  Config FICD = ICS_PGD1, JTAGEN_OFF
```

Note.

The **PLL_Setup** helper macro can be found within the device's ".def" file. It is required because the dsPIC33®, PIC24E® and PIC24H® devices need an unlock sequence before writing to the OSCCON SFR, unlike the PIC24F® devices, that can write directly to the OSCCON SFR.

DelayCs

Syntax
DelayCs *Length*

Overview
Delay execution for an amount of instruction cycles.

Operands
- Length can only be a constant with a value from 1 to 16383.

Example
```
DelayCs 100             ' Delay for 100 cycles
```

Note.
DelayCs is oscillator independent.

The length of a given instruction cycle is determined by the oscillator frequency divided by 2. The higher the oscillator, the smaller the cycle.

See also : DelayUs, **DelayMs**.

DelayMs

Syntax
DelayMs *Length*

Overview
Delay execution for *length* x milliseconds (ms). Delays may be up to 65535ms (65.535 seconds) long.

Operands
Length can be a constant, variable, or expression.

Example
```
Device = 24FJ64GA002
Declare Xtal = 16

Dim MyByte as Byte
Dim MyWord as Word

MyByte = 50
MyWord= 1000
DelayMs 100            ' Delay for 100ms
DelayMs MyByte         ' Delay for 50ms
DelayMs MyWord         ' Delay for 1000ms
DelayMs MyWord + 10    ' Delay for 1010ms
```

Note.
DelayMs is oscillator independent, as long as you inform the compiler of the crystal frequency to use, using the **Xtal** directive.

See also : Declare, DelayCs, DelayUs.

DelayUs

Syntax
DelayUs *Length*

Overview
Delay execution for *length* x microseconds (us). Delays may be up to 65535us (65.535 milli-seconds) long.

Operands
Length can be a constant, variable, or expression.

Example
```
Device = 24FJ64GA002
Declare Xtal = 16

Dim MyByte as Byte
Dim MyWord as Word

MyByte = 50
MyWord= 1000
DelayUs 1             ' Delay for 1us
DelayUs 100           ' Delay for 100us
DelayUs MyByte        ' Delay for 50us
DelayUs MyWord        ' Delay for 1000us
DelayUs MyWord + 10   ' Delay for 1010us
```

Note.
DelayUs is oscillator independent, as long as you inform the compiler of the crystal frequency to use, using the **Xtal** directive.

See also : **Declare, DelayUs, DelayMs.**

Device

Syntax
Device *Device name*

Overview
Inform the compiler which microcontroller is being used.

Operands
Device name can be any value PIC24E, PIC24F, PIC24H, dsPIC33F or dsPIC33E type.

Example

```
Device = 24FJ64GA002    ' Produce code for a 24FJ64GA002 device
```

Device should be the first directive placed in the program.

For an up-to-date list of compatible devices refer to the compiler's PPI folder.

Default location:

For Windows XP or Windows 7 32-bit:
C:\Program Files\ProtonIDE\PDS\Includes\PPI

For Windows 7 64-bit:
C:\Program Files (x86)\ProtonIDE\Includes\PPI

Dig

Syntax
Variable **= Dig** *Value, Digit number*

Overview
Returns the value of a decimal digit.

Operands
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 - 9 with 0 being the rightmost digit).

Example
```
Device = 24FJ64GA002
Declare Xtal = 16
Declare Hserial_Baud = 9600     ' USART1 baud rate
Declare Hrsout1_Pin = PORTB.14  ' Select the pin for TX with USART1

Dim MyByte as Byte
Dim Result as Byte

RPOR7 = 3                   ' Make PPS Pin RP14 U1TX

MyByte = 124
Result = Dig MyByte, 1      ' Extract the second digit's value
Hrsout Dec Result           ' Display the value, which is 2
```


Dim

Syntax
Dim *Variable* ***as*** *Size*

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

Operands
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**, **Double**, **String**, **Code**, or **PSV**

Example
```
' Declare different sized variables
  Dim MyByte as Byte          ' Declare an unsigned 8-bit Byte variable
  Dim MyWord as Word          ' Declare an unsigned 16-bit Word variable
  Dim MyDword as Dword        ' Declare an unsigned 32-bit Dword variable

  Dim sMyByte as SByte        ' Declare a signed 8-bit SByte variable
  Dim sMyWord as SWord        ' Declare a signed 16-bit SWord variable
  Dim sMyDword as SDword      ' Declare a signed 32-bit SDword variable

  Dim MyBit as Bit            ' Declare a 1-bit Bit variable
  Dim MyFloat as Float        ' Create a 32-bit floating point variable
  Dim MyDouble as Double      ' Create a 64-bit floating point variable

  Dim MyString 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
  Dim MyCode as PSV = 1,2,3,4,5,6,7 ' Place 7 bytes in PSV code memory
```

Notes.
Any RAM 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 MyByte as Byte
```
or
```
  Dim My_Byte as Word
```
or
```
  Dim My_Bit 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.
- **Double** Requires 8 bytes of RAM.
- **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 declared.

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

- **String** type variables can hold a maximum of 8192 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 2's 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's 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's 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.

- **Double** type variables may hold a value larger than **Float** types, and with some extra accuracy, 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 one of the most versatile of the variable family types. However, more so than **Dword** and **Float** types, this comes at a price because 64-bit 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**, **HighSByte**, **LowSByte**, **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 MyWord as Word            ' Declare an unsigned Word variable
   Dim MyWord_Hi as MyWord.HighByte
' MyWord_Hi now represents the unsigned high byte of variable MyWord
```

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

HighSByte 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 MyWord as SWord              ' Declare a signed Word variable
    Dim MyWord_Hi as MyWord.SByte1
' MyWord_Hi now represents the signed high byte of variable MyWord
```

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

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. **HighSByte** will still extract the signed high byte of the variable, as will **SByte3**.

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

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

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

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
```

Creating Code Memory Tables using Dim

There are two special cases of the Dim directive. These are:

```
Dim MyCode As Code
```

and

```
Dim MyCode As PSV
```

Both will create a data table in the device's code memory, however, the **PSV** directive will ensure that the **AddressOf** function returns the PSV address of the table, instead of its actual code memory address. This used mainly for DSP operations.

The data produced by the Code or PSV directives follows the same casting rules as the Cdata directive, in that the table's data can be given a size that each element will occupy.

```
Dim MyCode as Code = As Word 1, 2, 3, 4, 5
```

or

```
Dim MyCode as PSV = As Dword 100, 200, 300, 400
```

Note.
A code or PSV data table will not be included into a program if it is not used somewhere within the program.

Creating variables in Y RAM

dsPIC33® devices have an extra area of RAM dedicated to DSP operations. It resides at the top of the X RAM area and is named Y RAM. All DSP operations involving the accumulators must use Y RAM, otherwise the microcontroller will create an exception.

Adding the text **YRAM** at the end of a variable's declaration will cause it to be created in the Y RAM section:

```
Dim MyArray[10] As Word YRAM = 1, 2, 3, 4, 5, 6, 7
```

Each dsPIC33® family has differing amounts of YRAM, so the compiler will produce an error message if the limit is exceeded.

Creating variables in DMA RAM

Some PIC24® and dsPIC33® devices have an extra area of RAM dedicated to DMA (Direct Memory Access) operations. It resides at the top of the X RAM area, above any Y RAM, and is named DMA RAM. All DMA operations must use DMA RAM, otherwise the microcontroller will create an exception.

Adding the text **DM** at the end of a variable's declaration will cause it to be created in the DMA RAM section:

```
Dim MyArray[10] As Word DMA = 1, 2, 3, 4, 5, 6, 7
```

Each device has differing amounts of DMA RAM, if any, so the compiler will produce an error message if the limit is exceeded.

Notes.

The final RAM usage will also encompass the microcontroller's stack size, therefore, even if the BASIC program only declares 4 byte variables, the final RAM count will be 84. 80 bytes for the default stack size and 4 bytes for variable usage. If handled interrupts are used, the stack size will increase due to context saving and restoring requirements.

RAM locations for variables is allocated automatically within the microcontroller because the PIC24® and dsPIC33® range of devices have specific requirements concerning RAM addressing. Which are:

- 16-bit variables must be located on a 16-bit RAM address boundary.
- 32-bit and 64-bit variables must be placed on a 16-bit address boundary, but should be placed on a 32-bit address, if possible, for more efficiency with some mnemonics.
- 8-bit variables can be located on an 8-bit,16-bit or 32-bit RAM address boundary.

Therefore, the order of variable placements is:

- The microcontroller's 16-bit stack is located before all variables are placed.
- The compiler's 16-bit system variables are placed.
- **Word** variables are placed.
- **Dword** variables are placed.
- **Float** variables are placed.
- **Double** variables are placed.
- **Byte** variables are placed.
- **Word Arrays** are placed.
- **Dword Arrays** are placed.
- **Float Arrays** are placed.
- **Byte Arrays** are placed.
- **String** variables are placed.

The logic behind the variable placements is because of the microcontroller's near and far RAM.

The first 8192 bytes of RAM are considered "near" RAM, while space above that is considered "far" RAM. By default, the compiler sets all user variables to near RAM. However, when near RAM space is full, the compiler will place variables in far RAM (above 8192).

The special significance of near versus far to the compiler is that near RAM accesses are encoded in only one mnemonic using direct addressing, while accesses to variables in far RAM require two to three mnemonics using indirect addressing.

Standard variables are used more commonly within a BASIC program, therefore should reside in near RAM for efficiency. Arrays and Strings are generally accessed indirectly anyway, therefore, it is of little consequence if they reside in near or far RAM.

See Also : **Aliases, Declaring Arrays, Floating Point Math, Symbol, Creating and using Strings .**

DTMFout

Syntax
DTMFout *Pin*, { *OnTime* }, { *OffTime*, } **[** *Tone* {, *Tone*…} **]**

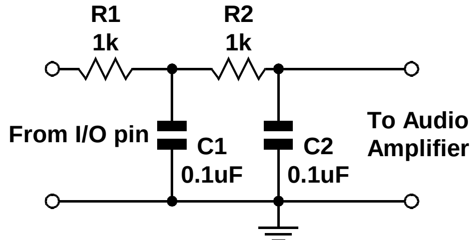
Overview
Produce a DTMF Touch Tone sequence on *Pin.*

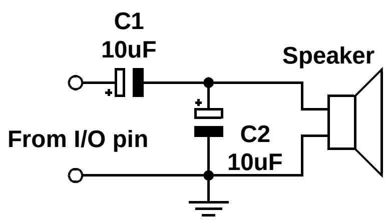
Operands
Pin is a Port.Bit constant that specifies the I/O pin to use. This pin will be set to output during generation of tones and set to input after the command is finished.
OnTime is an optional variable, constant, or expression (0 - 65535) specifying the duration, in ms, of the tone. If the *OnTime* 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 microcontroller was connected to the phone line correctly, the above command would dial 749990. 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 microcontroller 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 microcontroller 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 microcontroller 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 microcontroller'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 reference; 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 devices.

Operands
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 microcontroller 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 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 because they do not occupy code memory, but reside in a separate part of memory.

The example program below illustrates the use of eeprom addressing.

```
' Display two text strings held in eeprom memory

  Device = 24F08KL301
  Declare Xtal = 16
  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 things 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 at the beginning of the table will force a given length.

```
Edata as Dword 100000, 10000, 1000, 100, 10, 1
```

Byte will force the value to occupy one byte of eeprom 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 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.

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

  Device = 24F08KL301
  Declare Xtal = 16

  Dim P10 as Dword          ' Power of 10 variable
  Dim BCount as Byte
  Dim Index as Byte

  Dim Value as Dword        ' Value to convert
  Dim String1[11] as Byte ' Holds the converted value
  Dim Pointer as Byte       ' Pointer within the Byte array

  DelayMs 100               ' Wait for things 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:
  Pointer = 0
  Index = 0
  Repeat
    P10 = Eread Index * 4
    BCount = 0

    While Value >= P10
      Value = Value - P10
      Inc BCount
    Wend
    If BCount <> 0 Then
      String1[Pointer] = BCount + "0"
      Inc Pointer
    EndIf
    Inc Index
  Until Index > 8
  String1[Pointer] = Value + "0"
  Inc Pointer
  String1[Pointer] = 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 code memory tables
' Based on their address located in a separate table
'
  Device = 24F08KL301
  Declare Xtal = 16
'
' Table of address's located in eeprom memory
'
  Edata as Word String1, String2

  Dim DataByte as Byte
  Dim String1 as Code = "Hello",0
  Dim String2 as Code = "World",0

  WREG10 = Eread 0                ' Locate the address of the first string
  While                           ' Create an infinite loop
    DataByte = cPtr8(WREG10++)    ' Read each character from the code string
    If DataByte = 0 Then Break    ' Exit if null found
    Hrsout DataByte               ' Display the character
  Wend                            ' Close the loop
  Hrsout 13

  WREG10 = Eread 2                ' Locate the address of the second string
  While                           ' Create an infinite loop
    DataByte = cPtr8(WREG10++)    ' Read each character from the code string
    If DataByte = 0 Then Break   ' Exit if null found
    Hrsout DataByte               ' Display the character
  Wend                            ' Close the loop
  Hrsout 13
```

See also : Eread, Ewrite.

End

Syntax
End

Overview
The **End** statement creates an infinite loop.

Notes.
End stops the microcontroller processing by placing it into a continuous loop. The port pins remain the same.

See also : **Stop.**

Eread

Syntax
Variable **= Eread** *Address*

Overview
Read information from the on-board eeprom available on some devices.

Operands
Variable is a user defined variable.
Address is a constant, variable, or expression, that contains the address of interest within eeprom memory.

Example
```
Device = 24F08KL301
Declare Xtal = 16

Dim MyByte As Byte
Dim MyWord As Word
Dim MyDword As Dword

Edata 10, 354, 123456789      ' Place some data into the eeprom
MyByte = Eread 0              ' Read the 8-bit value from address 0
MyWord = Eread 1              ' Read the 16-bit value from address 1
MyDword = 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: -

```
MyWord.LowByte = Eread 0      ' Read an 8-bit value
MyWord.HighByte = 0           ' Clear the high byte of MyWord
```

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.

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 5ms per byte.

See also : **Edata, Ewrite**

Ewrite

Syntax
Ewrite *Address*, **[** *Variable* {, *Variable*…etc } **]**

Overview
Write information to the on-board eeprom available on some devices.

Operands
Address is a constant, variable, or expression, that contains the address of interest within eeprom memory.
Variable is a user defined variable.

Example
```
Device = 24F08KL301
Declare Xtal = 16

Dim MyByte as Byte
Dim MyWord as Word
Dim Address as Byte
MyByte = 200
MyWord = 2456
Address = 0                ' Point to address 0 within the eeprom
Ewrite Address, [MyWord, MyByte] ' 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, [MyWord.LowByte, MyByte]
```

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, [MyWord]
Next
```

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 5ms 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.

Operands
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 MyWord within an upward loop
  Device = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600     ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14  ' Select the pin for TX with USART1

  Dim MyWord as Word

  RPOR7 = 3                        ' Make PPS Pin RP14 U1TX
  For MyWord = 0 to 2000 Step 2    ' Perform an upward loop
    Hrsout Dec MyWord, 13          ' Display the value of MyWord
  Next                             ' Close the loop
```

Example 2
```
' Display in decimal, all the values of MyWord within a downward loop
  Device = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600     ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14  ' Select the pin for TX with USART1

  Dim MyWord as Word

  RPOR7 = 3                        ' Make PPS Pin RP14 U1TX
  For MyWord = 2000 to 0 Step -2   ' Perform a downward loop
    Hrsout Dec MyWord, 13          ' Display the value of MyWord
  Next                             ' Close the loop
```


Example 3

```
' Display in decimal, all the values of MyDword within a downward loop
  Device = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600      ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14  ' Select the pin for TX with USART1

  Dim MyDword as Dword

  RPOR7 = 3                         ' Make PPS Pin RP14 U1TX
  For MyDword = 200000 to 0 Step -200  ' Perform a downward loop
    Hrsout Dec MyDword, 13             ' Display the value of MyDword
  Next                                 ' Close the loop
```

Example 4

```
' Display all of MyWord1 using expressions as parts of the For-Next
  Device = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600      ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14  ' Select the pin for TX with USART1

  Dim MyWord1 as Word
  Dim MyWord2 as Word

  RPOR7 = 3                         ' Make PPS Pin RP14 U1TX
  MyWord2 = 1000

  For MyWord1= MyWord2 + 10 to MyWord2 + 1000   ' Perform a loop
    Hrsout Dec MyWord1, 13                ' Display the value of MyWord1
  Next                                    ' Close the loop
```

Notes.

It may have been noticed from the above examples, that no variable is present after the **Next** command. A variable name after **Next** is purely optional.

For-**Next** loops may be nested as deeply as the code memory on the microcontroller will allow. To break out of a loop you may use the **GoTo** command without any ill effects, which is exactly what the **Break** command does: -

```
  For MyByte = 0 to 20                 ' Create a loop of 21
    If MyByte = 10 Then GoTo BreakOut  ' Break out of loop when MyByte is 10
  Next                                 ' Close the loop

BreakOut:
```

See also : **While...Wend, Repeat...Until.**

Freqout

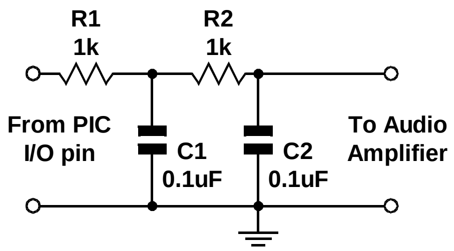
Syntax
Freqout *Pin, Period, Freq1* {, *Freq2*}

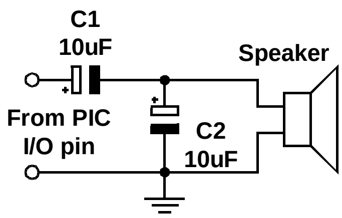
Overview
Generate one or two sine-wave tones, of differing or the same frequencies, for a specified period.

Operands
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 or 40MHz 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 = 24FJ64GA002
  Declare Xtal = 16

  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

  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.**

GetBit

Syntax
Variable **= GetBit** *Variable*, *Index*

Overview
Examine a bit of a variable, or register.

Operands
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 MyByte
  Device = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600      ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14   ' Select the pin for TX with USART1

  Dim MyByte as Byte
  Dim Index as Byte
  Dim Var1 as Byte

  RPOR7 = 3                         ' Make PPS Pin RP14 U1TX

  MyByte = %10110111
  While                             ' Create an infinite loop
    Hrsout Bin8 MyByte, 13          ' Display the original variable
    For Index = 7 to 0 Step -1    ' Create a loop for 8 bits
      Var1 = GetBit MyByte,Index    ' Examine each bit of MyByte
      Hrsout Dec1 Var1              ' Display the binary result
      DelayMs 100                   ' Slow things down to see what's happen-
ing
    Next                            ' Close the loop
    Hrsout 13
  Wend                              ' Do it forever
```

See also : **ClearBit, LoadBit, SetBit.**

Gosub

Syntax
Gosub *Label*

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.

Operands
Label is a user-defined label.

Example 1
```
' Implement a standard subroutine call
  GoTo Main         ' Jump over the subroutines
SubA:
  subroutine A code
  ……
  ……
  Return

SubB:
  subroutine B code
  ……
  ……
  Return

' Actual start of the main program
Main:
  Gosub SubA
  Gosub SubB
```

A subroutine must always end with a **Return** command.

GoTo

Syntax
GoTo *Label*

Overview
Jump to a defined label and continue execution from there.

Operands
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 I^{2}C bus using the microcontroller's MSSP module.

Notes.
Because of the subtleties involved in interfacing to some I^{2}C devices, the compiler's standard **Hbusin** and **Hbusout** commands were found lacking. Therefore, individual pieces of the I^{2}C 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 = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600      ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14   ' Select the pin for TX with USART1

  Dim Loop as Byte
  Dim Array[10] as Byte

  RPOR7 = 3                         ' Make PPS Pin RP14 U1TX
'
' 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
  Hrsout Str Array, 13       ' Display the Array as a String
```

See also : **HbusAck, HbRestart, HbStop, Hbusin, Hbusout.**

HbStop

Syntax
HbStop

Overview
Send a **Stop** condition to the I^2C bus using the microcontroller's MSSP module.

HbRestart

Syntax
HbRestart

Overview
Send a **Restart** condition to the I^2C bus using the microcontroller's MSSP module.

HbusAck

Syntax
HbusAck

Overview
Send an **Acknowledge** condition to the I^2C bus using the microcontroller's MSSP module.

HbusNack

Syntax
HbusNack

Overview
Send a **Not Acknowledge** condition to the I^2C 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 I^2C bus using the MSSP module, and places it into *variable/s*. If syntax structures *Two* or *Four* (see above) are used, then No Acknowledge, or Stop is sent after the data. Syntax structures *One* and *Three* first send the *control* and optional *address* out of the clock pin (*SCL*), and data pin (*SDA*).

Operands
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* syntax 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. HbStart, HbRestart, HbusAck, or HbStop. The *First*, and *Third* syntax types may be used to receive several values and designate each to a separate variable, or variable type.

The **Hbusin** command operates as an I^2C master, using the microcontroller's MSSP module, and may be used to interface with any device that complies with the 2-wire I^2C 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 MyByte 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
  MyByte = Hbusin Control, Address  ' Read the byte from the eeprom
```

or

```
  Hbusin Control, Address, [MyByte]  ' 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 MyWord as Word          ' Declare a Word size variable
  MyWord = Hbusin Control, Address
```

Will receive a 16-bit value from the bus. While: -

```
  Dim MyByte as Byte          ' Declare a Byte size variable
  MyByte = 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 MyByte as Byte
  Dim MyWord as Word
  Hbusin Control, Address, [MyByte, MyWord]
```

Will receive two values from the bus, the first being an 8-bit value dictated by the size of variable MyByte which has been declared as a byte. And a 16-bit value, this time dictated by the size of the variable MyWord 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* syntax variations allow all the subtleties of the I^2C 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 Declares

Declare HSDA_Pin Port **.** Pin
Declares the port and pin used for the data line (SDA). The location of the port and pin used for hardware I^2C can be altered by the fuse configurations. If the declare is not used in the program, it will default to the standard pin configuration.

Declare HSCL_Pin Port **.** Pin
Declares the port and pin used for the clock line (SCL). The location of the port and pin used for hardware I^2C can be altered by the fuse configurations. If the declare is not used in the program, it will default to the standard pin configuration.

Declare Hbus_Bitrate Constant 100, 400, 1000 etc.
The standard speed for the I^2C bus is 100KHz. Some devices use a higher bus speed of 400KHz. The above **Declare** allows the I^2C 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.
Because the I^2C 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 I^2C 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 MyArray[10] as Byte ' Define an array of 10 bytes
  Dim Address as Byte     ' Create a word sized variable

  Hbusin %10100000, Address, [Str MyArray] ' Load data into all the array
' Load data into only the first 5 elements of the array
  Hbusin %10100000, Address, [Str MyArray\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 MyArray      ' Load all the array with bytes received
  HbStop                  ' Send a Stop condition
```

An alternative ending to the above example is: -

```
  Hbusin Str MyArray\5  ' Load data into only the first 5 elements of 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 I^{2}C 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.

Operands
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 I^{2}C master and may be used to interface with any device that complies with the 2-wire I^{2}C 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 MyByte 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
  MyByte = 200                            ' The value place into address 20
  Hbusout Control, Address, [MyByte] ' 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 MyWord as Word          ' Declare a Word size variable
Hbusout Control, Address, [MyWord]
```

Will send a 16-bit value to the bus. While: -

```
Dim MyByte as Byte          ' Declare a Byte size variable
Hbusout Control, Address, [MyByte]
```

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 MyByte as Byte
Dim MyWord as Word
Hbusout Control, Address, [MyByte, MyWord]
```

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 MyWord 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", MyByte, MyWord]
```

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 I^2C 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 microcontroller's CARRY flag, which is SR.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
  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 SRbits_C = 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
```


Hbusout Declares

Declare HSDA_Pin Port **.** Pin
Declares the port and pin used for the data line (SDA). The location of the port and pin used for hardware I^2C can be altered by the fuse configurations. If the declare is not used in the program, it will default to the standard pin configuration.

Declare HSCL_Pin Port **.** Pin
Declares the port and pin used for the clock line (SCL). The location of the port and pin used for hardware I^2C can be altered by the fuse configurations. If the declare is not used in the program, it will default to the standard pin configuration.

Declare Hbus_Bitrate Constant 100, 400, 1000 etc.
The standard speed for the I^2C bus is 100KHz. Some devices use a higher bus speed of 400KHz. The above **Declare** allows the I^2C 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.
When the **Hbusout** command is used, the appropriate SDA and SCL Port and Pin are automatically setup as inputs. Because the I^2C 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 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.

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.

Operands
Port can be any valid port.
Port.Bit can be any valid port and bit combination, i.e. PORTB.1

Example
```
Device = 24HJ128GP502
Declare Xtal = 16

Symbol LED = PORTB.4
High LED

High PORTB
High PORTA.0
```

Note.
The compile will write to the device's LAT SFR and will always set the relevant Port or Port.Bit to an output.

See also : **Clear, Dim, Low, Set, Symbol.**

Hpwm

Syntax
Hpwm *Channel, Dutycycle, Frequency*

Overview
Output a pulse width modulated pulse train using on of the OCP modules. The PWM pulses produced can run continuously in the background while the program is executing other instructions.

Operands
Channel is a constant value that specifies which hardware PWM channel to use (1 to 5). must be the same on all channels. It must be noted, that this is a limitation of the devices not the compiler. The data sheet for the particular device used shows the fixed hardware pin for each Channel.
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 (0-65535), 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 65535Hz. The lowest usable **Hpwm** *Frequency* at each oscillator setting is dependant on the oscillator frequency that the device is operating with.

Example
```
  Device = 24FJ64GA002
  Declare Xtal = 16

  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
```

Notes.
Some devices 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 OCP1 module.
Declare CCP2_Pin Port.Pin    ' Select Hpwm port and bit for OCP2 module.
Declare CCP3_Pin Port.Pin    ' Select Hpwm port and bit for OCP3 module.
Declare CCP4_Pin Port.Pin    ' Select Hpwm port and bit for OCP4 module.
Declare CCP5_Pin Port.Pin    ' Select Hpwm port and bit for OCP5 module.
```

See also : **Pwm, Pulseout, Servo.**

Hrsin, Hrsin2, Hrsin3, Hrsin4

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.

Operands
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.
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 = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600      ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14   ' Select the pin for TX with USART1

  Dim MyByte as Byte

  RPOR7 = 3                       ' Make PPS Pin RP14 U1TX

  While
    MyByte = Hrsin, {1000, Timeout}  ' Receive a byte serially into MyByte
    Print Dec MyByte, " "         ' Display the byte received
  Wend                            ' Loop forever
Timeout:
  Cls
  Print "Timed Out"               ' Display an error if Hrsin timed out
```

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 microcontroller was 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 is 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 microcontroller 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 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 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 microcontroller, 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 **Hrsin** command only offers a 8 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 microcontroller will receive the data properly.

Declares

There are several Declare directives for use with the **Hrsin** commands. These are: -

Declare HRsin_Pin Port **.** Pin
Declares the port and pin used for USART1 reception (RX). The location of the port and pin is dictated by the device's PPS (Peripheral Pin Select) options. Note that this declare will not alter any PPS (Peripheral Pin Select) SFRs.

Declare Hserial_Baud Constant value
Sets the Baud rate that will be used to receive a value serially from USART1. 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 9600 baud.

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 (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.

The hardware serial ports (USARTs) only have a small input buffer, therefore, they can easily overflow if characters are not read from it often enough. When this occurs, USART1 stops accepting any new characters, and requires resetting. This overflow error can be reset by clearing the OERR bit within the U1STA register:

```
Clear U1STAbits_OERR  ' Clear an overflow error for USART1
```

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
```


Declare HRsin2_Pin Port . Pin
Declares the port and pin used for USART2 reception (RX). The location of the port and pin is dictated by the device's PPS (Peripheral Pin Select) options. Note that this declare will not alter any PPS (Peripheral Pin Select) SFRs.

Declare Hserial2_Baud Constant value
Sets the Baud rate that will be used to receive a value serially from USART2. 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 9600 baud.

Declare Hserial2_Parity Odd or Even
Enables/Disables parity on the serial port. For both **Hrsin2** and **Hrsout2** 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 **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.

The hardware serial ports (USARTs) only have a small input buffer, therefore, they can easily overflow if characters are not read from it often enough. When this occurs, USART2 stops accepting any new characters, and requires resetting. This overflow error can be reset by clearing the OERR bit within the U2STA register:

```
Clear U2STAbits_OERR  ' Clear an overflow error for USART2
```

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
```

Declare HRsin3_Pin Port . Pin
Declares the port and pin used for USART3 reception (RX). The location of the port and pin is dictated by the device's PPS (Peripheral Pin Select) options. Note that this declare will not alter any PPS (Peripheral Pin Select) SFRs.

Declare Hserial3_Baud Constant value
Sets the Baud rate that will be used to receive a value serially from USART3. 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 9600 baud.

Declare Hserial3_Parity Odd or Even
Enables/Disables parity on the serial port. For both **Hrsin3** and **Hrsout3** 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 **Hserial3_Parity** declare.

```
Declare Hserial3_Parity = Even   ' Use if even parity desired
Declare Hserial3_Parity = Odd    ' Use if odd parity desired
```

Declare Hserial3_Clear On or Off
Clear the overflow error bit before commencing a read.

The hardware serial ports (USARTs) only have a small input buffer, therefore, they can easily overflow if characters are not read from it often enough. When this occurs, USART3 stops accepting any new characters, and requires resetting. This overflow error can be reset by clearing the OERR bit within the U3STA register:

```
Clear U3STAbits_OERR  ' Clear an overflow error for USART3
```

Alternatively, the **Hserial3_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 Hserial3_Clear = On
```

Declare HRsin4_Pin Port **.** Pin
Declares the port and pin used for USART4 reception (RX). The location of the port and pin is dictated by the device's PPS (Peripheral Pin Select) options. Note that this declare will not alter any PPS (Peripheral Pin Select) SFRs.

Declare Hserial4_Baud Constant value
Sets the Baud rate that will be used to receive a value serially from USART4. 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 9600 baud.

Declare Hserial4_Parity Odd or Even
Enables/Disables parity on the serial port. For both **Hrsin4** and **Hrsout4** 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 **Hserial4_Parity** declare.

```
Declare Hserial4_Parity = Even    ' Use if even parity desired
Declare Hserial4_Parity = Odd     ' Use if odd parity desired
```

Declare Hserial4_Clear On or Off
Clear the overflow error bit before commencing a read.

The hardware serial ports (USARTs) only have a small input buffer, therefore, they can easily overflow if characters are not read from it often enough. When this occurs, USART4 stops accepting any new characters, and requires resetting. This overflow error can be reset by clearing the OERR bit within the U4STA register:

```
Clear U4STAbits_OERR  ' Clear an overflow error for USART4
```

Alternatively, the **Hserial4_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 Hserial4_Clear = On
```


Notes.
The **Hrsin** commands 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.

See also : Declare, Rsin, Rsout, Hrsout, Hserin, Hserout.

Hrsout, Hrsout2, Hrsout3, Hrsout4

Syntax
Hrsout *Item* {, *Item...* }

Overview
Transmit one or more *Ite*ms from a USART on devices that support asynchronous serial communications in hardware.

Operands
Item may be a constant, variable, expression, string list, or inline command.
There are no operands as such, instead there are *modifiers*. For example, if an at sign'@' precedes an *Ite*m, the ASCII representation for each digit is transmitted.

The modifiers are listed below: -

Modifier	Operation
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 Label	Send string data defined in code memory.

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 MyFloat as Float
MyFloat = 3.145
Hrsout Dec2 MyFloat    ' 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 MyFloat as Float
MyFloat = 3.1456
Hrsout Dec MyFloat    ' 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 MyFloat as Float
MyFloat = -3.1456
Hrsout Dec MyFloat    ' Send 3 digits after the decimal point
```

The above program will transmit the ASCII characters “-3.145”

Example 1

```
Dim MyByte as Byte
Dim MyWord as Word
Dim MyDword as Dword

Hrsout "Hello World\r"            ' Display the text "Hello World"
Hrsout "Var1= ", Dec MyByte, 13  ' Display the decimal value of MyByte
Hrsout "Var1= ", Hex MyByte, 13  ' Display the hexadecimal value of MyByte
Hrsout "Var1= ", Bin MyByte, 13  ' Display the binary value of MyByte
Hrsout "MyDword= ", Hex6 MyDword, 13 ' Display 6 hex characters
```

The **Cstr** modifier may be used in commands that deal with text processing i.e. **Serout**, **Hserout**, and **Print** etc. However, the **Cstr** keyword is not always required, because the compiler recognises a label name as a null terminated string of characters.

The **Cstr** modifier can be used in conjunction with code memory strings. The **Dim as Code** directive is used for initially creating the string of characters: -

```
Dim MyCodeString as Code = "Hello World", 0
```

The above line of case will create, in code memory, the values that make up the ASCII text "Hello World", at address MyCodeString. 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 MyCodeString
```

The label that declared the address where the list of code memory values resided, now becomes the code memory 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.

First the standard way of displaying text: -

```
Device = 24FJ64GA002
Declare Xtal = 16
Declare Hserial_Baud = 9600       ' USART1 baud rate
Declare Hrsout1_Pin = PORTB.14    ' Select the pin for TX with USART1

RPOR7 = 3                         ' Make PPS Pin RP14 U1TX

Hrsout "Hello World\r"
Hrsout "How are you?\r"
Hrsout "I am fine!\r"
```

Now using a code memory string: -

```
Dim Text1 as Code = "Hello World\r", 0
Dim Text2 as Code = "How are you?\r", 0
Dim Text3 as Code = "I am fine!\r", 0

Hrsout Text1
Hrsout Text2
Hrsout Text3
```

Again, note the null terminators after the ASCII text in the code memory strings. Without these, the device will continue to transmit data until it sees a value of 0.

Internally, the compiler is placing the quoted strings of characters into code memory, therefore either of the above constructs is valid, however, the compiler also internally combines quoted strings that are identical, meaning that the first of the above constructs can be more efficient in some cases.

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 microcontroller 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 several **Declare** directives for use with the **Hrsout** commands. These are: -

For HRSout

Declare HRsout_Pin Port **.** Pin

Declares the port and pin used for USART1 transmission (TX). The location of the port and pin is dictated by the device's PPS (Peripheral Pin Select) options. Note that this declare will not alter any PPS (Peripheral Pin Select) SFRs.

There is no default setting for this **Declare** and it must be used within the BASIC program.

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_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 (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 Hrsout_Pace 0 to 65535 microseconds (us)
Implements a delay between characters transmitted by the **Hrsout** 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 **Hrsout**.

If the **Declare** is not used in the program, then the default is no delay between characters.

For HRsout2

Declare HRsout2_Pin Port **.** Pin
Declares the port and pin used for USART2 transmission (TX). The location of the port and pin is dictated by the device's PPS (Peripheral Pin Select) options. Note that this declare will not alter any PPS (Peripheral Pin Select) SFRs.

There is no default setting for this **Declare** and it must be used within the BASIC program.

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.

Declare Hserial2_Parity Odd or Even
Enables/Disables parity on the serial port. For both **Hrsout2** and **Hrsin2** 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 **Hserial2_Parity** declare.

```
Declare Hserial2_Parity = Even     ' Use if even parity desired
Declare Hserial2_Parity = Odd      ' Use if odd parity desired
```

Declare Hrsout2_Pace 0 to 65535 microseconds (us)
Implements a delay between characters transmitted by the **Hrsout2** 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 **Hrsout2**.

If the **Declare** is not used in the program, then the default is no delay between characters.

For HRsout3

Declare HRsout3_Pin Port **.** Pin
Declares the port and pin used for USART3 transmission (TX). The location of the port and pin is dictated by the device's PPS (Peripheral Pin Select) options. Note that this declare will not alter any PPS (Peripheral Pin Select) SFRs.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare Hserial3_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.

Declare Hserial3_Parity Odd or Even
Enables/Disables parity on the serial port. For both **Hrsout3** and **Hrsin3** 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 **Hserial3_Parity** declare.

```
Declare Hserial3_Parity = Even      ' Use if even parity desired
Declare Hserial3_Parity = Odd       ' Use if odd parity desired
```

Declare Hrsout3_Pace 0 to 65535 microseconds (us)
Implements a delay between characters transmitted by the **HRsout3** 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 **Hrsout3**.

If the **Declare** is not used in the program, then the default is no delay between characters.

For HRsout4

Declare HRsout4_Pin Port **.** Pin
Declares the port and pin used for USART4 transmission (TX). The location of the port and pin is dictated by the device's PPS (Peripheral Pin Select) options. Note that this declare will not alter any PPS (Peripheral Pin Select) SFRs.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare Hserial4_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.

Declare Hserial4_Parity Odd or Even
Enables/Disables parity on the serial port. For both **Hrsout4** and **Hrsin4** 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 **Hserial4_Parity** declare.

```
Declare Hserial4_Parity = Even      ' Use if even parity desired
Declare Hserial4_Parity = Odd       ' Use if odd parity desired
```

Declare Hrsout4_Pace 0 to 65535 microseconds (us)
Implements a delay between characters transmitted by the **Hrsout4** 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 **Hrsout4**.

If the **Declare** is not used in the program, then the default is no delay between characters.

Notes.
The **Hrsout commands** 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.

See also : Declare, Rsin, Rsout, Serin, Serout, Hrsin, Hserin, Hserout.

Hserin, Hserin2, Hserin3, Hserin4

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)

Operands
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.
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 = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600      ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14   ' Select the pin for TX with USART1

  Dim MyByte as Byte

  RPOR7 = 3                        ' Make PPS Pin RP14 U1TX

  While
    Hserin 1000, Timeout, [MyByte] ' Receive a byte serially into MyByte
    Print Dec MyByte, " "          ' Display the byte received
  Wend                             ' Loop forever
Timeout:
  Cls
  Print "Timed Out"                ' Display an error if Hserin timed out
```

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 is 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 microcontroller 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 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 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 microcontroller, 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 **Hserin** command offers a 8 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 microcontroller will receive the data properly.

Declares
There are several Declare directives for use with the **Hserin** commands. These are the same declares as used by the **HRsin** commands

Notes.
The **Hserin commands** 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.

See also : Declare, Hserout, Hrsin, Hrsout, Rsin, Rsout.

Hserout, Hserout2, Hserout3, Hserout4

Syntax
Hserout [*Item* {, *Item...* }**]**

Overview
Transmit one or more *Ite*ms from the USART on devices that support asynchronous serial communications in hardware.

Operands
Item may be a constant, variable, expression, string list, or inline command.
There are no operands as such, instead there are *modifiers*. For example, if an at sign'@' precedes an *Ite*m, the ASCII representation for each digit is transmitted.

The modifiers are listed below: -

Modifier	**Operation**
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 Label	Send string data defined in code memory.

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 MyFloat as Float
MyFloat = 3.145
Hserout [Dec2 MyFloat]     ' 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 MyFloat as Float
MyFloat = 3.1456
Hserout [Dec MyFloat]   ' 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 MyFloat as Float
MyFloat = -3.1456
Hserout [Dec MyFloat]   ' Send 3 values after the decimal point
```

The above program will transmit the ASCII representation of -3.145

Example

```
Device = 24FJ64GA002
Declare Xtal = 16
Declare Hserial_Baud = 9600      ' USART1 baud rate
Declare Hrsout1_Pin = PORTB.14   ' Select the pin for TX with USART1

Dim MyByte as Byte
Dim MyWord as Word
Dim MyDword as Dword

RPOR7 = 3                        ' Make PPS Pin RP14 U1TX
Hserout ["Hello World"]          ' Display the text "Hello World"
Hserout ["Var1= ", Dec MyByte]   ' Display the decimal value of MyByte
Hserout ["Var1= ", Hex MyByte]   ' Display the hexadecimal value of MyByte
Hserout ["Var1= ", Bin MyByte]   ' Display the binary value of MyByte
' Display 6 hex characters of a Dword type variable
Hserout ["MyDword= ", Hex6 MyDword]
```

The **Cstr** modifier is used in conjunction with code memory strings. The **Dim as Code** directive is used for initially creating the string of characters: -

```
Dim String1 as Code = "Hello World", 0
```

The above line of case 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 code memory 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 = 24FJ64GA002
Declare Xtal = 16
Declare Hserial_Baud = 9600       ' USART1 baud rate
Declare Hrsout1_Pin = PORTB.14    ' Select the pin for TX with USART1

RPOR7 = 3                         ' Make PPS Pin RP14 U1TX

Hserout ["Hello World\r"]
Hserout ["How are you?\r"]
Hserout ["I am fine!\r"]
```

Now using the **Cstr** modifier: -

```
Dim Text1 as Code = "Hello World", 0
Dim Text2 as Code = "How are you?\r", 0
Dim Text3 as Code = "I am fine!\r", 0

Hserout [Cstr Text1]
Hserout [Cstr Text2]
Hserout [Cstr Text3]
```

Again, note the null terminators after the ASCII text in the code memory strings. Without these, the device will continue to transmit data until a value 0 is reached.

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 microcontroller 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 several Declare directives for use with the **Hserout** commands. These are the same declares as used by the **HRsout** commands.

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.

See also : Declare, Rsin, Rsout, Serin, Serout, Hserin, Hserin.

I2Cin

Syntax
I2Cin *Dpin, Cpin, Control,* { *Address* }, **[** *Variable* {, *Variable*…} **]**

Overview
Receives a value from the I^2C bus, and places it into *Variable/s*.

Operands
Dpin is a Port.Pin constant that specifies the I/O pin that will be connected to the I^2C 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 I^2C device's clock line (SCL). This pin's I/O direction will be changed to input and will remain in that state after the instruction is completed.
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 I^2C master, and may be used to interface with any device that complies with the 2-wire I^2C 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.
  Device = 24FJ64GA002
  Declare Xtal = 16

  Dim MyByte 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 SSL (Clock) line

  Address = 20                   ' Read the value at address 20
  I2Cin SDA, SCL, Control, Address, [MyByte] ' 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 MyWord as Word
I2Cin SDA, SCL, Control, Address, [Var1, MyWord]
```

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 MyWord 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 I^2C 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 I^2C 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: -

```
  Device = 24FJ64GA002
  Declare Xtal = 16

  Dim Array[10] as Byte             ' Create 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 I^{2}C bus, by first sending the *control* and optional *address*.

Operands
Dpin is a Port.Pin constant that specifies the I/O pin that will be connected to the I^{2}C 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 I^{2}C 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 I^{2}C master and may be used to interface with any device that complies with the 2-wire I^{2}C 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.
  Device = 24FJ64GA002
  Declare Xtal = 16

  Dim MyByte 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 SSL (Clock) line
  Address = 20                    ' Write to address 20
  MyByte = 200                   ' The value place into address 20
  I2Cout SDA, SCL, Control, Address, [MyByte] ' 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 MyWord as Word              ' Declare a Word size variable
I2Cout SDA, SCL, Control, Address, [MyWord]
```

Will send a 16-bit value to the bus. While: -

```
Dim MyByte as Byte             ' Declare a Byte size variable
I2Cout SDA, SCL, Control, Address, [MyByte]
```

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 MyByte as Byte
Dim MyWord as Word
I2Cout SDA, SCL, Control, Address, [MyByte, MyWord]
```

Will send two values to the bus, the first being an 8-bit value dictated by the size of variable MyByte which has been declared as a byte. And a 16-bit value, this time dictated by the size of the variable MyWord 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", MyByte, MyWord]
```

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: -

```
Device = 24FJ64GA002
Declare Xtal = 16

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 I^2C 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
Eliminates the necessity for a pull-up resistor on the SCL line.

The I^2C 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 **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 pull-up 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 I^2C 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.

You may imagine that it's limiting having a fixed set of pins for the I^2C 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 microcontroller 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.

Operands
Comparison is composed of variables, numbers and comparators.
Instruction is the statement to be executed should the *comparison* fulfil the **If** criteria

Example 1
```
Device = 24FJ64GA002
Declare Xtal = 16
Symbol LED = PORTB.4
MyByte = 3
Low LED
If MyByte > 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 MyVar1 & 1 = 0 Then
    MyVar2 = 0
    MyVar3 = 1
  Else
    MyVar2 = 1
  EndIf
  If MyVar4 = 1 Then
    MyVar2 = 0
    MyVar3 = 0
  EndIf
```

Example 3
```
  If MyVar1 = 10 Then
    High LED1
  ElseIf MyVar1 = 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 MyVar1 = 10 Then High LED1 : ElseIf MyVar1 = 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 MyVar1 is equal to 10 then LED1 will illuminate, if MyVar1 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 if fulfilled: -

```
  If MyVar1 = 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 Proton24 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 Operands, 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.

Operands
Filename is any valid Proton24 file.

Example
```
' Main Program Includes sub files
  Include "StartCode.inc"
  Include "MainCode.inc"
  Include "EndCode.inc"
```

Notes.
The file to be included into the BASIC listing may be in one of several 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 user's Includes folder, located in the user's PDS directory.
4… Within the Includes folder of the compiler's current directory.
5… 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.
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. Var1 = Var1 + 1

Operands
Variable is a user defined variable

Example
```
Device = 24FJ64GA002
Declare Xtal = 16
Declare Hserial_Baud = 9600      ' USART1 baud rate
Declare Hrsout1_Pin = PORTB.14   ' Select the pin for TX with USART1

Dim MyDword as Dword

RPOR7 = 3                       ' Make PPS Pin RP14 U1TX
MyDword = 1
Repeat
  Hrsout Dec MyDword, 13
  DelayMs 200
  Inc MyDword
Until MyDword > 10000
```

The above example shows the equivalent to the For-Next loop: -

```
For MyDword = 1 to 10000 : Next
```

However, the **Repeat-Until** version, although it looks more complex, is much more efficient in both code size and speed of operation.

See also : Dec.

Inkey

Syntax
Variable **= Inkey**

Overview
Scan a keypad and place the returned value into *variable*

Operands
Variable is a user defined variable

Example
```
Device = 24FJ64GA002
Declare Xtal = 16
Declare Hserial_Baud = 9600      ' USART1 baud rate
Declare Hrsout1_Pin = PORTB.14  ' Select the pin for TX with USART1
Declare KeyPad_Port = PORTB

Dim MyByte as Byte
RPOR7 = 3                       ' Make PPS Pin RP14 U1TX
While                           ' Create an infinite loop
  MyByte = Inkey                ' Scan the keypad
  DelayMs 50                    ' Simple debounce by waiting 50ms
  Hrsout Dec MyByte, 13         ' Display the result
Wend                            ' Do it forever
```

Notes.
Inkey will return a value between 0 and 16. If no key is pressed, the value returned is 16.

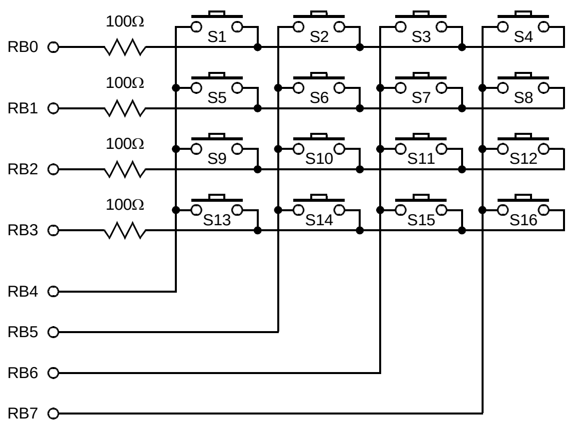
Using a **LookUp** command, the returned values can be re-arranged to correspond with the legends printed on the keypad: -

```
MyByte = Inkey
MyKey = LookUp MyByte, [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 it's connection configuration.

Declare

Declare Keypad_Port Port
Assigns the 8-bits of a Port that the keypad is attached to.



The diagram above illustrates typical connections for a 16-button keypad.

Input

Syntax
Input *Port* **.** *Pin*

Overview
Makes the specified *Port* or *Pin* an input.

Operands
Port.Pin must be a Port, or Port.Pin constant declaration.

Example
```
Input PORTB.0   ' Make bit-0 of PORTB an input

Input PORTB       ' Make all of PORTB an input
```

Notes.
An Alternative method for making a particular pin an input is by directly modifying the TRIS register: -

```
TRISB.0 = 1       ' Make bit-0 of PORTB an input
```

All of the pins on a port may be set to inputs by setting the whole TRIS register at once: -

```
TRISB = %1111111111111111 ' 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.**

Isr, EndIsr

Syntax
Isr *Interrupt Name, {UnHandled}*
Interrupt handler BASIC code goes here
EndIsr

Overview
Indicate the start and end of an interrupt handling subroutine.

Operands
Interrupt Name is the name of the interrupt being handled by the subroutine.
Unhandled is an optional parameter that will disable context saving and restoring of the WREG SFRs and key SFRs, as well as compiler system variables. Use this option with caution and only when you know that the interrupt handler's code will not disturb any other SFR or variable.

Unlike 8-bit PIC® microcontroller's, PIC24® and dsPIC33® devices have a separate vector for each type of interrupt. Each interrupt has a specific name, and there are up to 128 of them. A typical interrupt vector name list is shown below:

Interrupt Name	Interrupt Cause	
OscillatorFail	Oscillator Failure	(Non-Maskable)
StackError	Address Error	(Non-Maskable)
AddressError	Stack Error	(Non-Maskable)
MathError	Math Error	(Non-Maskable)
DMACError	DMA Error	(Non-Maskable)
INT0Interrupt	External Interrupt 0	
IC1Interrupt	Input Capture 1	
OC1Interrupt	Output Compare 1	
T1Interrupt	Timer1	
DMA0Interrupt	DMA Channel 0	
IC2Interrupt	Input Capture 2	
OC2Interrupt	Output Compare 2	
T2Interrupt	Timer2	
T3Interrupt	Timer3	
SPI1ErrInterrupt	SPI1 Error	
SPI1Interrupt	SPI1 Transfer Done	
U1RXInterrupt	UART1 Receiver	
U1TXInterrupt	UART1 Transmitter	
ADC1Interrupt	ADC 1	
DMA1Interrupt	DMA Channel 1	
SI2C1Interrupt	I^2C1 Slave Events	
MI2C1Interrupt	I^2C1 Master Events	
CNInterrupt	Change Notification Interrupt	
INT1Interrupt	External Interrupt 1	
IC7Interrupt	Input Capture 7	
IC8Interrupt	Input Capture 8	
DMA2Interrupt	DMA Channel 2	
OC3Interrupt	Output Compare 3	
OC4Interrupt	Output Compare 4	

Interrupt Name	Interrupt Cause
T4Interrupt	Timer4
T5Interrupt	Timer5
INT2Interrupt	External Interrupt 2
U2RXInterrupt	UART2 Receiver
U2TXInterrupt	UART2 Transmitter
SPI2ErrInterrupt	SPI2 Error
SPI2Interrupt	SPI1 Transfer Done
DMA3Interrupt	DMA Channel 3
IC3Interrupt	Input Capture 3
IC4Interrupt	Input Capture 4
IC5Interrupt	Input Capture 5
IC6Interrupt	Input Capture 6
OC5Interrupt	Output Compare 5
OC6Interrupt	Output Compare 6
OC7Interrupt	Output Compare 7
OC8Interrupt	Output Compare 8
DMA4Interrupt	DMA Channel 4
T6Interrupt	Timer6
T7Interrupt	Timer7
SI2C2Interrupt	I^2C2 Slave Events
MI2C2Interrupt	I^2C2 Master Events
T8Interrupt	Timer8
T9Interrupt	Timer9
INT3Interrupt	External Interrupt 3
INT4Interrupt	External Interrupt 4
U1ErrInterrupt	UART1 Error
U2ErrInterrupt	UART2 Error

There are commonalities for the names on all PIC24® and dsPIC33® devices, however, interrupt vector names will be added if the device has a specific peripheral. A full list of the interrupt names for a specific device can be found within the device's PPI file, under the [ISRSTART] section. The PPI files can be found within the compiler's "PDS\Includes\PPI" directory.

The first 5 names in the list are interrupts that cannot be disabled. i.e. Non-Maskable, and are used for exception handling within the microcontroller. The others in the list are Maskable, meaning they can be enabled or disabled accordingly.

The interrupt handler, unless otherwise indicated, will first disable any other interrupts, then save key SFRs (Special Function Registers) such as SR (STATUS on 8-bit devices), CORCON, RCOUNT, WREG0 to WREG14, then if available on the device being used, PSVPAG, DSRPAG, DCOUNT, DOSTART, and DOEND. If the device has more than 65K or code memory, the TBLPAG SFR will also be saved. Saving is accomplished by pushing the SFRs or variables onto the microcontroller's stack, which will expand to accommodate them.

It will then save any compiler system variables that are used within the interrupt handler, before re-enabling interrupts and handing control to the code within the interrupt handler. The reverse is accomplished when the interrupt is exited, and the **Retfie** mnemonic issued. Note that the interrupt handler will not reset any associated interrupt flag.

This means that pretty much any BASIC code can be placed inside an interrupt as long as it handles any peripheral conflicts, such as Port re-use etc...

If additional SFRs or variables need to be saved within the interrupt handler, they can by Pushed onto the stack then Popped from it before the interrupt exits. For example:

```
  Isr T1Interrupt
    Push TRISB                         ' Save 16-bit TRISB on the stack
    Push PORTB                         ' Save 16-bit PORTB on the stack
    Toggle PORTB                       ' Use TRISB and PORTB
    Pop PORTB                          ' Restore 16-bit PORTB from the stack
    Pop TRISB                          ' Restore 16-bit TRISB from the stack
    IFS0bits_T1IF = 0                  ' Reset the Timer1 interrupt flag
  EndIsr                               ' Exit the interrupt
```

Note that the compiler can only track its system variable use when the code is between **Isr** and **EndIsr**. It cannot track any code that is called as a subroutine or a procedure from the ISR handler.

An asm **Bra** mnemonic is placed before the **Isr** directive, that jumps over the handler code past the **EndIsr** directive. This means that the interrupt handler can be placed in the line of code without having to jump over it manually. However, this behaviour can be altered by adding a dash after the **Isr** and **EndIsr** directives: **Isr-** and **EndIsr-**. This will save two bytes of code space for every interrupt handler used in the BASIC program, but measures should be taken to make sure that the program does not run the interrupt code directly, such as a **GoTo** to the main program loop.

Example

```
'
' Timer interrupt demo
'
    Device = 24FJ64GA002
    Declare Xtal = 16
    Declare Hserial_Baud = 9600       ' UART1 baud rate
    Declare Hrsout1_Pin = PORTB.14    ' Select which pin for TX with USART1

    Dim FloatOut1 As Float
    Dim FloatOut2 As Float

'-------------------------------------------------------------------------
    GoTo Main                          ' Jump over the interrupt handlers
'-------------------------------------------------------------------------
' Timer1 interrupt handler
' Transmit a floating point value serially
'
Isr- T1Interrupt                       ' Context save
    HRSOut "Timer1 ", Dec1 FloatOut1, 13
    FloatOut1 = FloatOut1 + 0.1
    IFS0bits_T1IF = 0                  ' Reset the Timer1 interrupt flag
EndIsr-                                ' Context restore and exit the interrupt
'-------------------------------------------------------------------------
' Timer2 interrupt handler
' Transmit a floating point value serially
'
Isr - T2Interrupt                      ' Context save
    HRSOut "Timer2 ", Dec1 FloatOut2, 13
    FloatOut2 = FloatOut2 + 0.1
    IFS0bits_T2IF = 0                  ' Reset the Timer2 interrupt flag
EndIsr-                                ' Context restore and exit the interrupt
```



```
'-------------------------------------------------------------------------
Main:
    RPOR7 = 3                          ' Make PPS Pin RP14 U1TX
    FloatOut1 = 0
    FloatOut2 = 0
'
' Configure Timer1
'
    TMR1 = 0
    PR1 = 8192                       ' Load Timer1 period
    T1CON = %1010000000101001        ' Start Timer1
                                     ' Discontinue operation in Idle mode
                                     ' Gated time accumulation disabled
                                     ' 1:64 prescaler
                                     ' Do not synchronise external clock input
                                     ' Internal clock

    IFS0bits_T1IF = 0                ' Clear Timer1 interrupt flag
    IPC0bits_T1IP0 = 0               ' Set priority
    IEC0bits_T1IE = 1                ' Enable the Timer1 interrupt
'
' Configure Timer2
'
    TMR2 = 0
    PR2 = 8192                       ' Load Timer2 period
    T2CON = %1010000000110101        ' Start Timer2
                                     ' Discontinue operation in Idle mode
                                     ' Gated time accumulation disabled
                                     ' 1:256 prescaler
                                     ' Timer2 as 16-bit timer
                                     ' Internal clock

    IFS0bits_T2IF = 0                ' Clear Timer2 interrupt flag
    IPC1bits_T2IP0 = 0               ' Set priority
    IEC0bits_T2IE = 1                ' Enable the Timer2 interrupt
```

The program above shows a worse case scenario of each interrupt calculating and displaying floating point variables, which are among the most processor intensive operations.

Adding another interrupt is as simple as placing **Isr** and **EndIsr** directives with a given name, and configuring the microcontroller to initiate the interrupt. The compiler will take care of the rest as much as it can.

Notes.
Nesting of **Isr** and **EndIsr** directives is not allowed.

The naming of the interrupts is taken from the official Microchip™ documentation and are the names used by the Linker application.

LCDread

Syntax
Variable **= LCDread** *Ypos, Xpos*

Overview
Read a byte from a graphic LCD.

Operands
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 = 24HJ128GP502
  Declare Xtal = 16
'
' LCD interface pin assignments
'
  Declare LCD_Type = Samsung   ' Setup for a Samsung KS0108 graphic LCD
  Declare LCD_DTPort = PORTB.Byte0
  Declare LCD_CS1Pin = PORTB.8
  Declare LCD_CS2Pin = PORTB.9
  Declare LCD_ENPin = PORTB.10
  Declare LCD_RSPin = PORTB.11
  Declare LCD_RWPin = PORTB.12

  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
```

Notes.
The graphic LCDs that are compatible with Proton24 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.

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.

Operands
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 = 24HJ128GP502
  Declare Xtal = 16
'
' LCD interface pin assignments
'
  Declare LCD_Type = Samsung   ' Setup for a Samsung KS0108 graphic LCD
  Declare LCD_DTPort = PORTB.Byte0
  Declare LCD_CS1Pin = PORTB.8
  Declare LCD_CS2Pin = PORTB.9
  Declare LCD_ENPin = PORTB.10
  Declare LCD_RSPin = PORTB.11
  Declare LCD_RWPin = PORTB.12

  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
```

Example 2

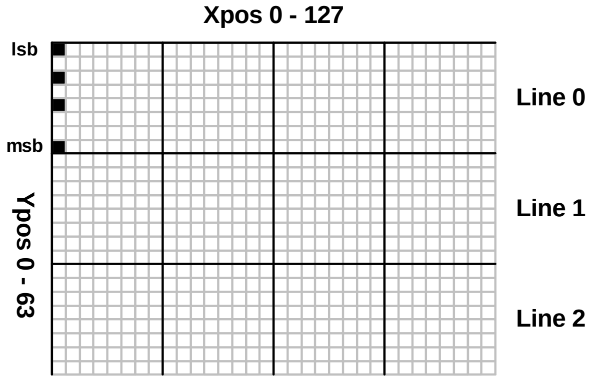
```
' Display a line on the top row of a Toshiba 128x64 graphic LCD
  Device = 24HJ128GP502
  Declare Xtal = 16

  Include "T6963C.Inc"            ' Load the T6983 routines into the pro-
gram

  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
```

Notes.

The graphic LCDs that are compatible with Proton24 are the Samsung KS0108, and the Toshiba T6963 (which must be included separately). 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.

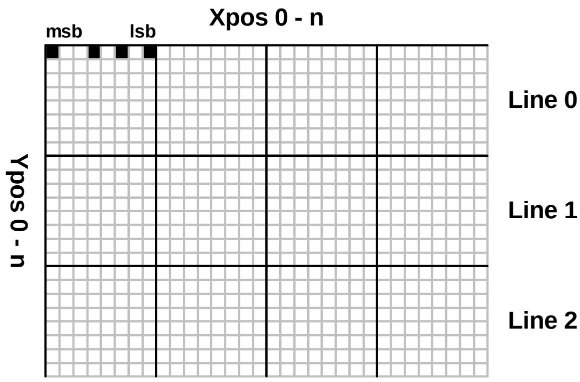
There are important differences between the Samsung and Toshiba screen formats. The diagrams below show these in more detail: -

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).



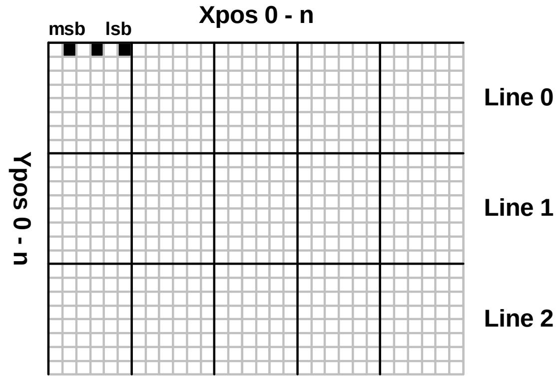
Samsung KS0108 graphic LCD

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).



Toshiba T6963 LCD. (8-bit Font mode)

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.



Toshiba T6963 LCD. (6-bit Font mode)

See also : **LCDread, Plot, Toshiba_Command, Toshiba_UDG, UnPlot see Print for LCD connections.**

Len

Syntax
Variable **= Len(***Source String***)**

Overview
Find the length of a **String**. (not including the null terminator) .

Operands
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 code memory.

Example 1
```
' Display the length of SourceString
  Device = 24HJ128GP502
  Declare Xtal = 16

  Dim SourceString as String * 20  ' Create a String capable of 20 charac-
ters
  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
```

Example 2
```
' Display the length of a Quoted Character String
  Device = 24HJ128GP502
  Declare Xtal = 16

  Dim Length as Byte

  Length = Len("Hello World")      ' Find the length
  Print Dec Length                 ' Display the result, which will be 11
```

Example 3
```
' Display the length of SourceString using a pointer to SourceString
  Device = 24HJ128GP502
  Declare Xtal = 16

  Dim SourceString as String * 20  ' Create a String capable of 20 charac-
ters
  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
```

Example 4

```
' Display the length of a code memory string
  Device = 24HJ128GP502
  Declare Xtal = 16

  Dim Length as Byte
'
' Create a null terminated string of characters in code memory
'
  Dim Source as Code = "Hello World", 0

  Length = Len(Source)        ' Find the length
  Print Dec Length            ' Display the result, which will be 11
```

See also : **Creating and using Strings, Creating and using code memory strings, 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.

Operands
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 = 24HJ128GP502
  Declare Xtal = 16

  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"
```

Example 2.

```
' Copy 5 chars from the left of a Quoted Character String into DestString
'
  Device = 24HJ128GP502
  Declare Xtal = 16

  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"
```

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 = 24HJ128GP502
  Declare Xtal = 16

  Dim SourceString as String * 20  ' Create a String capable of 20 charac-
ters
  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 code memory.

Example 4.

```
' Copy 5 characters from the left of a code memory table into DestString
'
  Device = 24HJ128GP502
  Declare Xtal = 16

  Dim DestString as String * 20  ' Create a String capable of 20 characters
' Create a null terminated string of characters in code memory
  Dim Source as Code = "Hello World", 0

' Copy 5 characters from label Source into the destination string
  DestString = Left$ (Source, 5)
  Print DestString               ' Display the result, which will be "Hello"
```

See also : **Creating and using Strings, Creating and using code memory strings, Len, Mid$, Right$, Str$, ToLower, ToUpper , AddressOf .**

Line

Syntax
Line *Pixel Colour*, *Xpos Start*, *Ypos Start*, *Xpos End*, *Ypos End*

Overview
Draw a straight line in any direction on a graphic LCD.

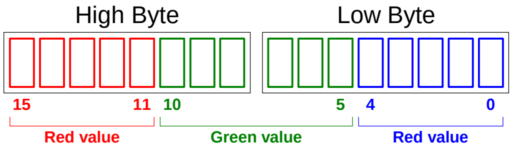
Operands
Pixel Colour may be a constant or variable that determines if the line will set or clear the pixels. A value of 1 will set the pixels and draw a line, while a value of 0 will clear any pixels and erase a line. If using a colour graphic LCD, this parameter holds the 16-bit colour of the pixel.
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 the LCD's X resolution.
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 the LCD's Y resolution.
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 the LCD's X resolution.
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 the LCD's Y resolution.

KS0108 graphci LCD example
```
' Draw a line from 0,0 to 120,34
'
  Device = 24HJ128GP502
  Declare Xtal = 16
'
' LCD interface pin assignments
'
  Declare LCD_Type = Samsung   ' Setup for a Samsung KS0108 graphic LCD
  Declare LCD_DTPort = PORTB.Byte0
  Declare LCD_CS1Pin = PORTB.8
  Declare LCD_CS2Pin = PORTB.9
  Declare LCD_ENPin = PORTB.10
  Declare LCD_RSPin = PORTB.11
  Declare LCD_RWPin = PORTB.12

  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 things 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
```

ILI9320 colour graphic LCD example

```
' Demonstrate the Line and LineTo commands with a colour LCD
'
  Device = 24EP128MC202
  Declare Xtal = 140.03
'
' Setup the Pins used by the ILI9320 graphic LCD
'
  Declare LCD_DTPort = PORTB.Byte0     ' Use the first 8-bits of PORTB
  Declare LCD_CSPin = PORTB.8          ' Connect to the LCD's CS pin
  Declare LCD_RDPin = PORTB.9          ' Connect to the LCD's RD pin
  Declare LCD_RSPin = PORTB.10         ' Connect to the LCD's RS pin
  Declare LCD_WRPin = PORTA.3          ' Connect to the LCD's WR pin

  Include "ILI9320.inc"        ' Load the ILI9320 routines into the program

  Dim wXpos As Word                    ' Create a variable for the X position
  Dim wYpos As Word                    ' Create a variable for the Y position

'---------------------------------------------------------------------------
Main:
' Configure the Oscillator to operate the device at 140.03MHz
'
  PLL_Setup(76, 2, 2, $0300)

  Cls clWhite                          ' Clear the LCD with the colour white
'
' Draw a series of lines
'
  For wYpos = 0 To 319
    Line clBrightBlue,0,0,239,wYpos
  Next
  For wYpos = 0 To 319
    Line clBrightRed,239,0,0,wYpos
  Next
'
' Draw a box around the LCD using LineTo
'
  DelayMS 512
  Line clBlack,1,1,238,1
  LineTo clBlack,238,318
  LineTo clBlack,1,318
  LineTo clBlack,1,1
'---------------------------------------------------------------------------
' Configure for internal 7.37MHz oscillator with PLL
' OSC pins are general purpose I/O
'
  Config FGS = GWRP_OFF, GCP_OFF
  Config FOSCSEL = FNOSC_FRCPLL, IESO_ON, PWMLOCK_OFF
  Config FOSC = POSCMD_NONE, OSCIOFNC_ON, IOL1WAY_OFF, FCKSM_CSDCMD
  Config FWDT = WDTPOST_PS256, WINDIS_OFF, PLLKEN_ON, FWDTEN_OFF
  Config FPOR = ALTI2C1_ON, ALTI2C2_OFF
  Config FICD = ICS_PGD1, JTAGEN_OFF
```

Notes.
With an ILI9320 colour graphic LCD, the colour is a 16-bit value formatted in RGB565, where the upper 5-bits represent the red content, the middle 6-bits represent the green content, and the lower 5-bits represent the blue content. As illustrated below:



For convenience, there are several colours defined within the ILI9320.inc file. These are:

```
clBlack
clBrightBlue
clBrightGreen
clBrightCyan
clBrightRed
clBrightMagenta
clBrightYellow
clBlue
clGreen
clCyan
clRed
clMagenta
clBrown
clLightGray
clDarkGray
clLightBlue
clLightGreen
clLightCyan
clLightRed
clLightMagenta
clYellow
clWhite
```

More constant values for colours can be added by the user if required.

See Also : **Box, Circle, LintTo, Plot.**

LineTo

Syntax
LineTo *Pixel Colour*, *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.

Operands
Pixel Colour 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. If using a colour graphic LCD, this parameter holds the 16-bit colour of the pixel.
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 the LCD's X resolution.
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 the LCD's Y resolution.

Example
```
' Draw a line from 0,0 to 120,34. Then from 120,34 to 0,63
  Device = 24HJ128GP502
  Declare Xtal = 16
'
' LCD interface pin assignments
'
  Declare LCD_Type = Samsung   ' Setup for a Samsung KS0108 graphic LCD
  Declare LCD_DTPort = PORTB.Byte0
  Declare LCD_CS1Pin = PORTB.8
  Declare LCD_CS2Pin = PORTB.9
  Declare LCD_ENPin = PORTB.10
  Declare LCD_RSPin = PORTB.11
  Declare LCD_RWPin = PORTB.12
  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 things 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
```

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, Plot.

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.

Operands
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 = 24HJ128GP502
  Declare Xtal = 16

  Dim ExVar as Word
  Dim Index as Byte
  Dim Value as Byte
  Dim PT_Var as Word

  While
    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
  Wend                            ' 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. 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, 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 *variabl*e. If no match is found, then the *variable* is unaffected.

Operands
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
```
Device = 24HJ128GP502
Declare Xtal = 16

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:

```
Device = 24HJ128GP502
Declare Xtal = 16

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 : Dim, cPtr8, cPtr16, cPtr32, Cread8, Cread16, Cread32, Edata, Eread, LookDownL, LookUp, LookUpL.

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 *value*1; 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 *variabl*e, or until all entries are exhausted, in which case *variable* is unaffected.

Operands
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 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 65536 values may be placed between the square brackets.
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 MyWord, [ 512, MyWord1, 1024 ]
Var1 = LookDownL MyWord, < [ 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 : **Dim, cPtr8, cPtr16, cPtr32, Cread8, Cread16, Cread32, Edata, Eread, LookDown, LookUp, LookUpL.**

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.

Operands
Variable may be a constant, variable, or expression. This is where the retrieved value will be stored.
Index may be a constant of 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 65536 values may be placed between the square brackets.

Example
```
' Create an animation of a spinning line.
  Device = 24HJ128GP502
  Declare Xtal = 16

  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 : **Dim, cPtr8, cPtr16, cPtr32, Cread8, Cread16, Cread32, Edata, Eread, LookDown, LookDownL, LookUpL.**

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.

Operands
Variable may be a constant, variable, or expression. This is where the retrieved value will be stored.
Index may be a constant of 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 65536 values may be placed between the square brackets.

Example
```
Device = 24HJ128GP502
Declare Xtal = 16

Dim Var1 as Byte
Dim MyWord as Word
Dim Index as Byte
Dim Assign as Word
Var1 = 10
MyWord = 1234
Index = 0        ' Point to the first value in the list (MyWord)
Assign = LookUpL Index, [MyWord, 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 : **Dim, cPtr8, cPtr16, cPtr32, Cread8, Cread16, Cread32, Edata, Eread, LookDown, LookDownL, LookUp.**

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.

Operands
Port can be any valid port.
Port.Bit can be any valid port and bit combination, i.e. PORTA.1

Example
```
Device = 24HJ128GP502
Declare Xtal = 16

Symbol LED = PORTB.4
Low LED
Low PORTB.0      ' Clear PORTB bit 0
Low PORTB        ' Clear all of PORTB
```

Note.
The compile will write to the device’s LAT SFR and will always set the relevant Port or Port.Bit to an output.

See also : **Dim, High, Symbol.**

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.

Operands
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 = 24HJ128GP502
  Declare Xtal = 16

  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"
```

Example 2
```
' Copy 5 chars from position 4 of a Quoted Character String into DestString

  Device = 24HJ128GP502
  Declare Xtal = 16

  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 "
```

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 = 24HJ128GP502
  Declare Xtal = 16

  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 "
```

A third possibility for *Source String* is a Label name, in which case a null terminated Quoted String of Characters is read from code memory.

Example 4

```
' Copy 5 characters from position 4 of a code memory string into DestString

  Device = 24HJ128GP502
  Declare Xtal = 16

  Dim DestString as String * 20    ' Create a String of 20 characters
' Create a null terminated string of characters in code memory
  Dim Source as Code = Hello World", 0

' 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"
```

See also : **Creating and using Strings, Creating and using code memory strings, 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.

Operands
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.

Example

```
  Device = 24HJ128GP502
  Declare Xtal = 16

  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.

See also : Branch, BranchL, 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.

Operands
Index Variable is a constant, variable, or expression, that specifies the label to call.
Label1...***Labeln*** are valid labels that specify where to call.

Example
```
  Device = 24HJ128GP502
  Declare Xtal = 16

  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..

See also : Branch, BranchL, On GoTo.

Output

Syntax
Output *Port or Port* **.** *Pin*

Overview
Makes the specified *Port* or *Port.Pin* an output.

Operands
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         ' Make bit-0 of PORTB an output
```

All of the pins on a port may be set to output by setting the whole TRIS register at once: -

```
TRISB = %0000000000000000    ' 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.**

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.

Operands
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
```
Device = 24HJ128GP502
Declare Xtal = 16

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).

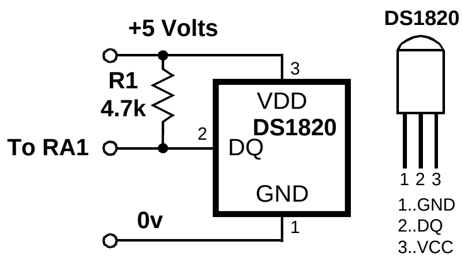
The reset pulse is controlled by the lowest two bits of the Mode argument in the Oread command. It can be made to appear before the ROM Function Command (Mode = 1), after the Transaction / Data portion (Mode = 2), before and after the entire transaction (Mode = 3) or not at all (Mode = 0).

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 micro-controller 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 microcontroller will ultimately have to address them individually using the Match ROM command.

The third part, the Memory Function Command, allows the microcontroller 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 = 24HJ128GP502
  Declare Xtal = 16

  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.

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 d vices.

Operands
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 control's 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 microcontroller 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, or if using a colour graphic LCD, it will hold the 16-bit colour of the pixel.

Operands
Variable is a user defined variable that holds the colour of the pixel.
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 = 24HJ128GP502
  Declare Xtal = 16
'
' KS0108 graphic LCD declares
'
  Declare LCD_Type = Samsung    ' Setup for a Samsung KS0108 graphic LCD
  Declare LCD_DTPort = PORTB.Byte0
  Declare LCD_CS1Pin = PORTB.8
  Declare LCD_CS2Pin = PORTB.9
  Declare LCD_ENPin = PORTB.10
  Declare LCD_RSPin = PORTB.11
  Declare LCD_RWPin = PORTB.12

  Dim Xpos as Byte
  Dim Ypos as Byte
  Dim Result as Byte

  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
```

See also : LCDread, LCDwrite, Plot, UnPlot. See Print for circuit.

Plot

Syntax
Plot *Ypos, Xpos*

Overview
Set an individual pixel on a graphic LCD.

Operands
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.

KS0108 LCD example
```
  Device = 24HJ128GP502
  Declare Xtal = 16
'
' KS0108 graphic LCD declares
'
  Declare LCD_Type = Samsung    ' Setup for a Samsung KS0108 graphic LCD
  Declare LCD_DTPort = PORTB.Byte0
  Declare LCD_CS1Pin = PORTB.8
  Declare LCD_CS2Pin = PORTB.9
  Declare LCD_ENPin = PORTB.10
  Declare LCD_RSPin = PORTB.11
  Declare LCD_RWPin = PORTB.12

  Dim Xpos as Byte
'
' 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
```

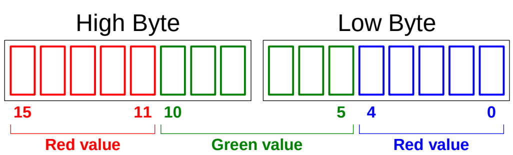
ILI9320 colour graphic LCD example

```
' Fill the LCD with colour using plot
'
  Device = 24EP128MC202
  Declare Xtal = 140.03
'
' Setup the Pins used by the ILI9320 320x240 pixel graphic LCD
'
  Declare LCD_DTPort = PORTB.Byte0    ' Use the first 8-bits of PORTB
  Declare LCD_CSPin = PORTB.8         ' Connect to the LCD's CS pin
  Declare LCD_RDPin = PORTB.9         ' Connect to the LCD's RD pin
  Declare LCD_RSPin = PORTB.10        ' Connect to the LCD's RS pin
  Declare LCD_WRPin = PORTA.3         ' Connect to the LCD's WR pin

  Include "ILI9320.inc"           ' Load the ILI9320 routines into the program

  Dim wXpos As Word             ' Create a variable for the X position
  Dim wYpos As Word               ' Create a variable for the Y position
'-------------------------------------------------------------------------
Main:
' Configure the internal oscillator to operate the device at 140.03MHz
  PLL_Setup(76, 2, 2, $0300)

  Cls clYellow                    ' Clear the LCD with the colour yellow
  Glcd_InkColour(clBrightBlue)    ' Choose the pixel colour
' Fill the LCD with colour
'
  For wYpos = 0 To 319
    For wXpos = 0 To 239
      Plot wYpos, wXpos
    Next
  Next
'-------------------------------------------------------------------------
' Configure for internal 7.37MHz oscillator with PLL
' OSC pins are general purpose I/O
'
  Config FGS = GWRP_OFF, GCP_OFF
  Config FOSCSEL = FNOSC_FRCPLL, IESO_ON, PWMLOCK_OFF
  Config FOSC = POSCMD_NONE, OSCIOFNC_ON, IOL1WAY_OFF, FCKSM_CSDCMD
  Config FWDT = WDTPOST_PS256, WINDIS_OFF, PLLKEN_ON, FWDTEN_OFF
  Config FPOR = ALTI2C1_ON, ALTI2C2_OFF
  Config FICD = ICS_PGD1, JTAGEN_OFF
```

Notes.

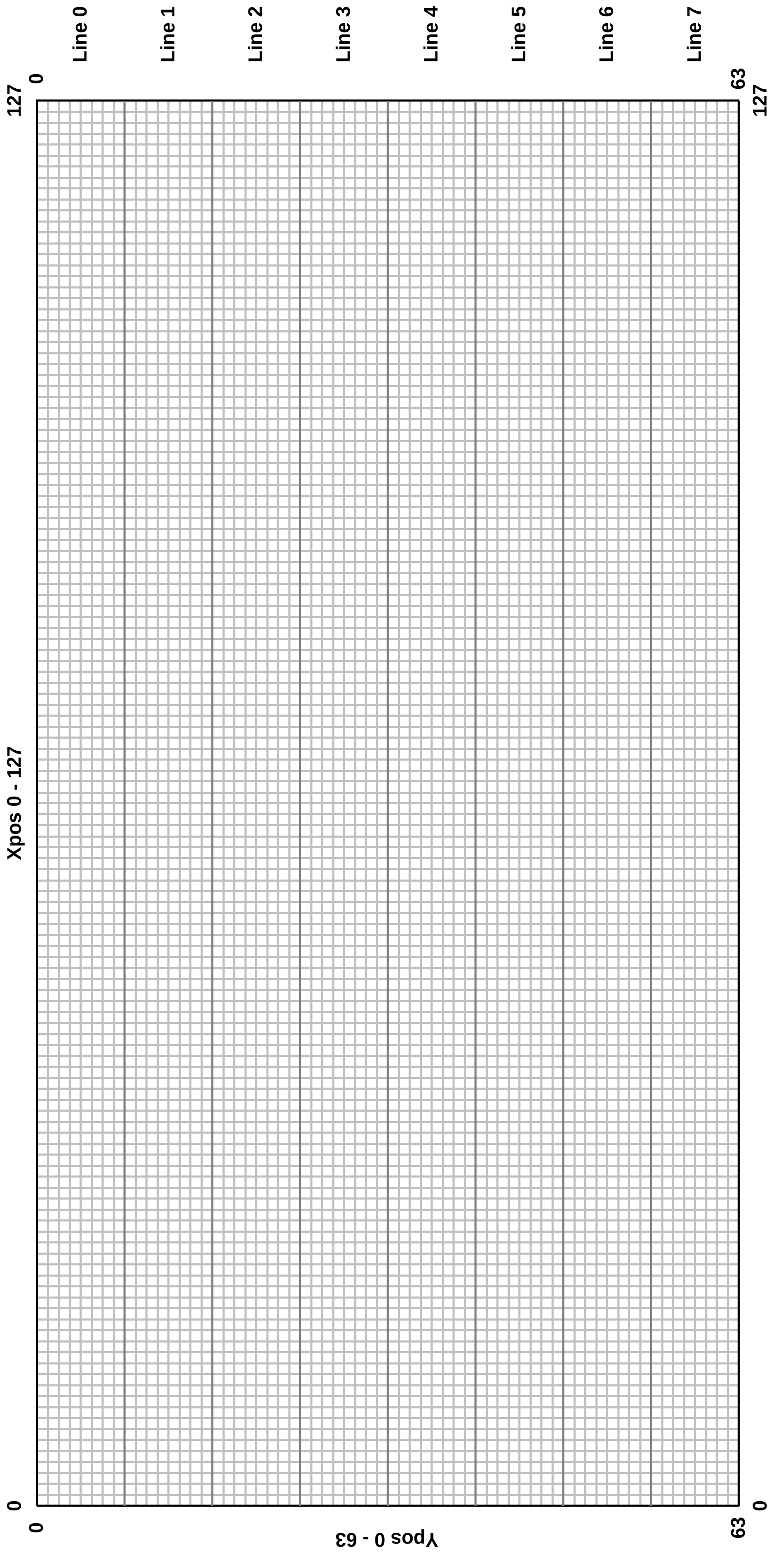
If using a colour graphic LCD, the **Plot** command will use the current colour of the pixel's Ink. As previously set by the **Glcd_Ink** command.

With an ILI9320 320x240 pixel colour graphic LCD, the colour is a 16-bit value formatted in RGB565, where the upper 5-bits represent the red content, the middle 6-bits represent the green content, and the lower 5-bits represent the blue content. As illustrated below:



See also : **LCDread, LCDwrite, Pixel, UnPlot.**

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 the microcontroller's stack.

Operands
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. The microcontroller's stack is word orientated, therefore all operations are accomplished using 16-bits.

Bit 2 Bytes are popped containing the value of the bit pushed.
Byte 2 Bytes are popped containing the value of the byte pushed.
Byte Array 2 Bytes are 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.
Float 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 = 24HJ128GP502
  Declare Xtal = 16
  Declare Stack_Size = 90    ' Increase the stack to hold extra words

  Dim MyWord as Word         ' Create a Word variable
  Dim MyDword as Dword       ' Create a Dword variable

  MyWord = 1234             ' Load the Word variable with a value
  MyDword = 567890           ' Load the Dword variable with a value
  Push MyWord, MyDword       ' Push the Word variable then the Dword variable

  Clear MyWord               ' Clear the Word variable
  Clear MyDword             ' Clear the Dword variable

  Pop MyDword, MyWord        ' Pop the Dword variable then the Word variable
  Print Dec MyWord, "  ", Dec MyDword ' Display the variables as decimal
```

Example 2
```
' Push a String on to the stack then retrieve it

  Device = 24HJ128GP502
  Declare Xtal = 16
  Declare Stack_Size = 90    ' Increase the stack to hold extra words

  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"
```

Example 3
```
' Push a Quoted character string on to the stack then retrieve it

  Device = 24HJ128GP502
  Declare Xtal = 16
  Declare Stack_Size = 90    ' Increase the stack to hold extra words

  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"
```

Notes.
Unlike the 8-bit PIC® microcontroller's, the PIC24® and dsPIC33® types have a true stack that occupied RAM and stores call and return data as well as data pushed onto it. This is a valuable resource for saving and restoring variables or SFRs than would otherwise be altered.

There are two declares for use with the stack. These are:

Declare Stack_Size = 20 to n (in words)
The compiler sets the default size of the microcontroller's stack to 60 words (120 bytes). This can be increased or decreased as required, as long as it fits within the RAM available. The compiler places a minimum limit of 20 for stack size. If the stack overflows or underflows, the microcontroller will trigger an exception.

Declare Stack_Expand = 1 or 0 or On or Off
Whenever an interrupt handler is used within a BASIC program, it must context save and restore critical SFRs and variables that would otherwise get overwritten. It uses the microcontroller's stack for temporary storage of the SFRs and variables, therefore the stack will increase with every interrupt handler used within the program. If this behaviour is undesirable, the above declare will disable it. However, the user must make sure that the stack is large enough to accommodate the storage, otherwise an exception will be triggered by the microcontroller.

See also : **Push.**

Pot

Syntax
Variable **= Pot** *Pin*, *Scale*

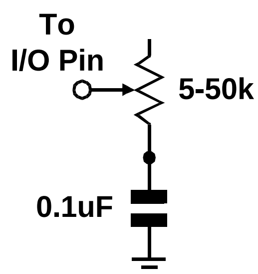
Overview
Read a potentiometer, thermistor, photocell, or other variable resistance.

Operands
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
```
Device = 24HJ128GP502
Declare Xtal = 16

Dim Var1 as Byte
While                         ' Create an infinite loop
  Var1 = Pot PORTB.0, 100     ' Read potentiometer on pin 0 of PORTB.
  Print Dec Var1, " "         ' Display the potentiometer reading
Wend                          ' Do it forever
```

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, or ILI9320 chipsets.

Operands
Item may be a constant, variable, expression, modifier, or string list.
There are no operands as such, instead there are *modifiers*. For example, if an at sign'@' precedes an *It*em, 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
Cls	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
Ibin{1..32}	Display binary digits with a preceding '%' identifier
Idec{1..10}	Display decimal digits with a preceding '#' identifier
Ihex{1..8}	Display hexadecimal digits with a preceding '$' identifier
ISbin{1..32}	Display signed binary digits with a preceding '%' identifier
ISdec{1..10}	Display signed decimal digits with a preceding '#' identifier
IShex{1..8}	Display signed hexadecimal digits with a preceding '$' identifier
Rep c\n	Display character c repeated n times
Str array\n	Display all or part of an array
Cstr Label	Display string data defined in code memory.

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 MyFloat as Float
MyFloat = 3.145
Print Dec2 MyFloat    ' 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 MyFloat as Float
MyFloat = 3.1456
Print Dec MyFloat   ' 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 MyFloat as Float
MyFloat = -3.1456
Print Dec MyFloat   ' 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

```
Device = 24HJ128GP502
Declare Xtal = 16
Dim Var1 as Byte
Dim MyWord as Word
Dim MyDword 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 "MyDword= ", Hex6 MyDword  ' 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.

The **Cstr** modifier is used in conjunction with code memory strings. The **Dim as Code** directive is used for initially creating the string of characters: -

```
Dim CodeString as Code = "Hello World", 0
```

The above line of case will create, in code 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 CodeString
```

The label that declared the address where the list of code memory values resided, now becomes the string's name.

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 microcontroller 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 microcontroller 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 microcontroller.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_ENPin Port **.** Pin
Assigns the Port and Pin that the LCD's EN line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_RSPin Port **.** Pin
Assigns the Port and Pins that the LCD's RS line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_Interface 4 or 8
Inform the compiler as to whether a 4-line or 8-line interface is required by the LCD.

There is no default setting for this **Declare** and it must be used within the BASIC program.

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.

There is no default setting for this **Declare** and it must be used within the BASIC program.

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
```

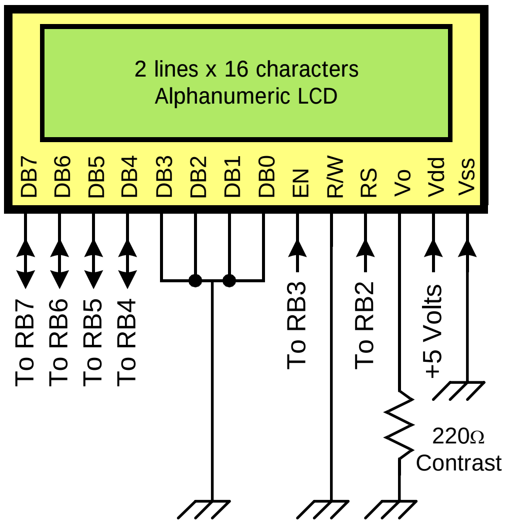
Will move the cursor to line 1, position 1 (HOME).

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 10
```



The above diagram shows typical connections for an alphanumeric LCD module using a 4-bit interface. Note that the compiler does not use the LCD's RW pin, and this must be connected to ground.

Using a KS0108 Graphic LCD

Once a KS0108 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: -

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
```

KS0108 graphic LCD Declares

There are several declares associated with a Samsung graphic LCD.

Declare LCD_DTPort Port = Port.Byte*n*

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_RWPin = Port **.** Pin

Assigns the Port and Pin that the graphic LCD's RW line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_ENPin Port **.** Pin

Assigns the Port and Pin that the LCD's EN line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_RSPin Port **.** Pin

Assigns the Port and Pins that the LCD's RS line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_CS1Pin = Port **.** Pin
Assigns the Port and Pin that the graphic LCD's CS1 line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare LCD_CS2Pin = Port **.** Pin
Assigns the Port and Pin that the graphic LCD's CS2 line will attach to.

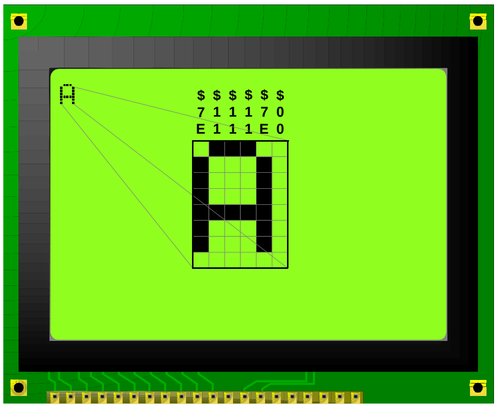
There is no default setting for this **Declare** and it must be used within the BASIC program.

Note
The KS0108 graphic LCD is a "non-intelligent" type, therefore, a separate character set is required. This is held internally in code memory.

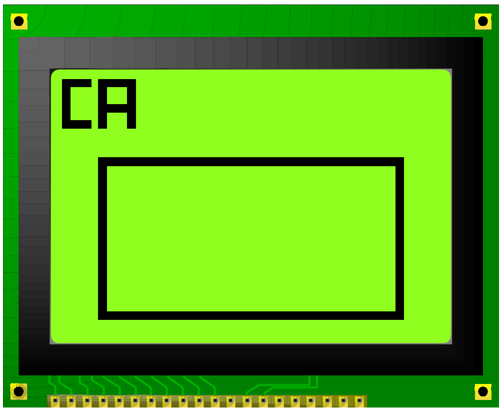
The code memory table that contains the font must have a label named **Font_Table**. For example: -

```
'{ data for characters 0 to 64 here}
  Dim Font_Table as Code =  7E, $11, $11, $11, $7E, $00,_  ' Chr 65 "A"
                            $7F, $49, $49, $49, $36, $00,_ ' Chr 66 "B"
  { rest of font table }
```

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.

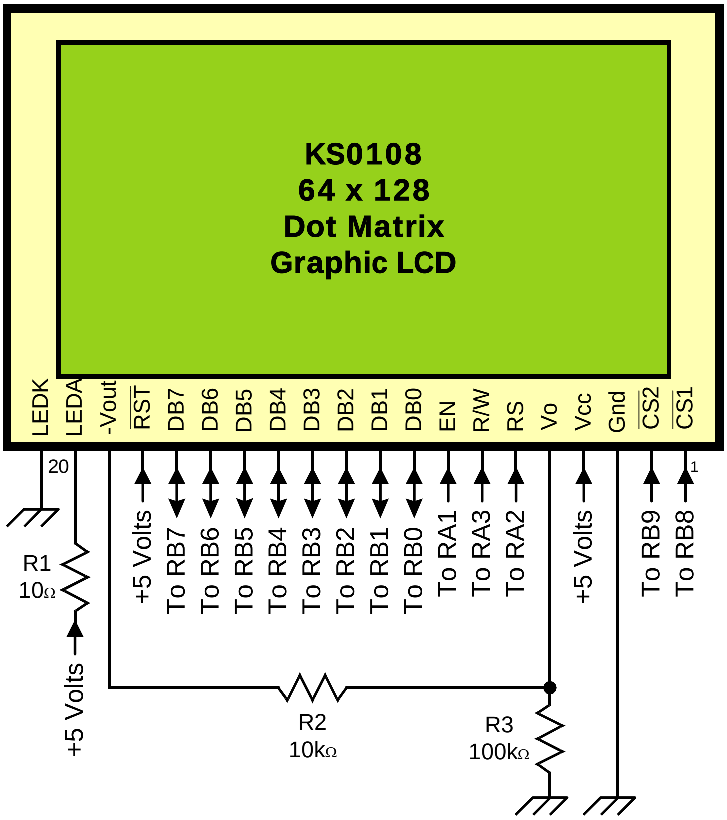


The character set itself is 128 characters long (0 -127). Which means that all the ASCII characters are present, including $, %, &, # etc.

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 16383 cycles.
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 the cycles delay is determined by the oscillator used.



The diagram above shows typical connections to a Samsung KS0108 graphic LCD.

Using a Toshiba T6963 Graphic LCD

Once a Toshiba T6963 graphic LCD has been chosen using the **Declare LCD_Type** directive, all **Print** outputs will be directed to that LCD. Note that the Toshiba routines must be loaded into the program via the **Include** directive.

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 **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 = Port.Byte*n*
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 subroutines.

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.

There is no default setting for this **Declare** and it must be used within the BASIC program.

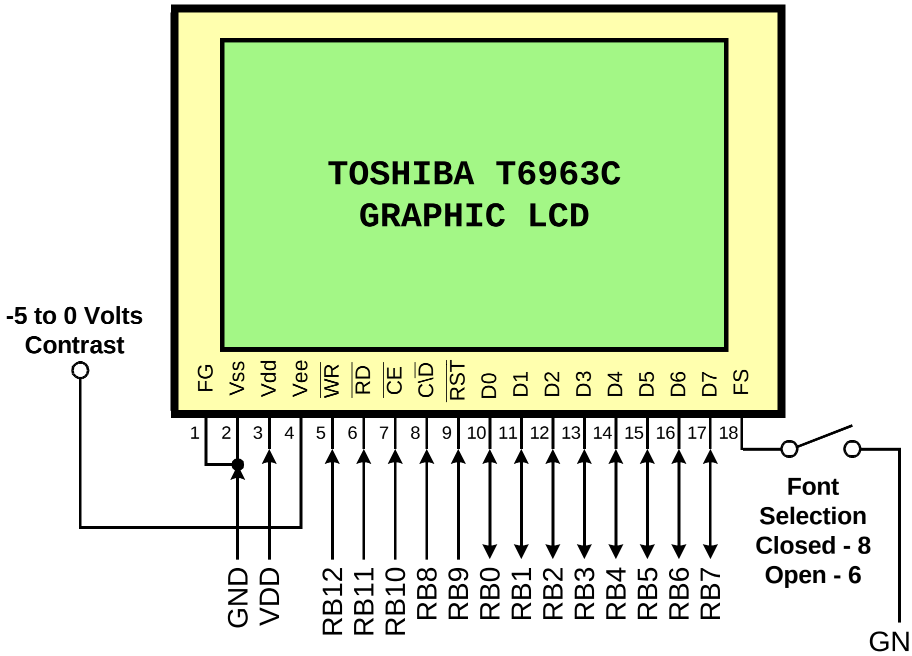
Declare LCD_Text_Pages 1 to n
As mentioned above, Toshiba graphic LCDs contain RAM that is set aside for text, graphics or characters generation. In normal use, only one page of text is all that is required, however, the compiler can re-arrange its library subroutines to allow several pages of text that 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. There is no default setting for this **Declare.**

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. There is no default setting for this **Declare.**

The diagram below shows a typical circuit for an interface with a Toshiba T6963 graphic LCD.



Example

```
' Toshiba T6963C graphic LCD demo
    Device = 24FJ64GA002
    Declare Xtal = 16
' Toshiba T6963C graphic LCD Pin configuration
    Declare LCD_Type = Toshiba        ' LCD's type is Toshiba T6963C
    Declare LCD_DTPort = PORTB.Byte0 ' The LCD's 8-bit Data port
    Declare LCD_WRPin = PORTB.12      ' The LCD's WR pin
    Declare LCD_RDPin = PORTB.11      ' The LCD's RD pin
    Declare LCD_CEPin = PORTB.10      ' The LCD's CE pin
    Declare LCD_CDPin = PORTB.8       ' The LCD's CD pin
    Declare LCD_RSTPin = PORTB.9      ' The LCD's RST pin (optional)
' Toshiba T6963C graphic LCD setup configuration
    Declare LCD_Font_Width = 8        ' The font width ( 6 or 8)
    Declare LCD_X_Res = 128           ' The X resolution of the LCD
    Declare LCD_Y_Res = 64            ' The Y resolution of the LCD
    Declare LCD_Text_Home_Address = 0  ' The home address of the LCD
    Declare LCD_RAM_Size = 8192       ' The amount of RAM the LCD contains
    Declare LCD_Text_Pages = 1          ' The amount of text pages required

    Include "T6963C.inc"  ' Load the Toshiba T6963C routines into the pro-
gram
' Create variables used in the demo
'
    Dim MyCodeString As Code = "From Code",0
    Dim MyRAMString As String * 20 = "From RAM"
    Dim MyWord As Word = 1234
    Dim MyDword As Dword = 12345
    Dim MyFloat As Float = 3.14

    Cls                                 ' Clear Text and Graphic RAM
    Print At 1,0, "1234567890ABCDEF"
    Print At 2,0, Dec MyWord
    Print At 3,0, Dec MyDword
    Print At 4,0, Dec MyFloat
    Print At 5,0, MyRAMString
    Print At 6,0, MyCodeString
```


Using an ILI9320 320x240 pixel Colour Graphic LCD

Once a colour 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 previous 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. And each cursor position is pixel based. The **Inverse**, **Xor**, and **Or** modifiers are not available for a colour LCD.

There are two new modifiers specifically for a colour LCD. These are: -

Ink 0 to 65535	Choose the colour of the pixel used for a character
Paper 0 to 65535	Choose the colour of the background under a character

Once one of the new modifiers has been enabled, all future **Print** commands will use that particular feature until the modifier is altered, or the Ink or Paper colour is chosen by the **GLCD_Ink** or **GLCD_Paper** commands. For example: -

```
' Enable red characters from this point
  Print At 0, 0, Ink cBrightRed, "Hello World"
  Print At 30, 0, "Still Red"
' Now use black characters
  Print At 60, 0, Ink cBlack, "Black 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
```

The routines for the ILI9320 colour graphic LCD must be included into the program before they are used, and the declares, listed later, must be placed. The LCD routines themselves are written in Proton24 BASIC so their operation may be readily changed.

Example.

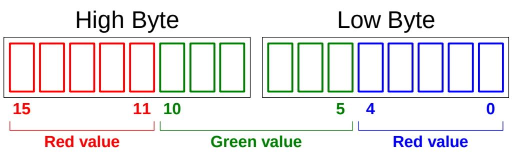
```
' Demonstrate the Print command with an ILI9320 colour graphic LCD
'
  Device = 24EP128MC202
  Declare Xtal = 140.03
'
' Setup the Pins used by the ILI9320 graphic LCD
'
  Declare LCD_DTPort = PORTB.Byte0   ' Use the first 8-bits of PORTB
  Declare LCD_CSPin = PORTB.8        ' Connect to the LCD's CS pin
  Declare LCD_RDPin = PORTB.9        ' Connect to the LCD's RD pin
  Declare LCD_RSPin = PORTB.10       ' Connect to the LCD's RS pin
  Declare LCD_WRPin = PORTA.3        ' Connect to the LCD's WR pin

  Include "ILI9320.inc"       ' Load the ILI9320 routines into the program
'-------------------------------------------------------
Main:
' Configure the internal oscillator to operate the device at 140.03MHz
  PLL_Setup(76, 2, 2, $0300)

  Cls clWhite                   ' Clear the LCD with the colour white
```



```
  Glcd_SetFont(CourierNew_20)      ' Choose the font to use
  Print At 0,0, Ink clBrightBlue, "Hello World"

'--------------------------------------------------------
' Load the font required
  Include "CourierNew_20.inc"
'
' Configure for internal 7.37MHz oscillator with PLL
' OSC pins are general purpose I/O
'
  Config FGS = GWRP_OFF, GCP_OFF
  Config FOSCSEL = FNOSC_FRCPLL, IESO_ON, PWMLOCK_OFF
  Config FOSC = POSCMD_NONE, OSCIOFNC_ON, IOL1WAY_OFF, FCKSM_CSDCMD
  Config FWDT = WDTPOST_PS256, WINDIS_OFF, PLLKEN_ON, FWDTEN_OFF
  Config FPOR = ALTI2C1_ON, ALTI2C2_OFF
  Config FICD = ICS_PGD1, JTAGEN_OFF
```

Fonts can be created using the supplied program named *FontCreator.exe*, and can be found in the IDE's *plugins* folder.

With an ILI9320 320x240 pixel colour graphic LCD, the colour is a 16-bit value formatted in RGB565, where the upper 5-bits represent the red content, the middle 6-bits represent the green content, and the lower 5-bits represent the blue content. As illustrated below:

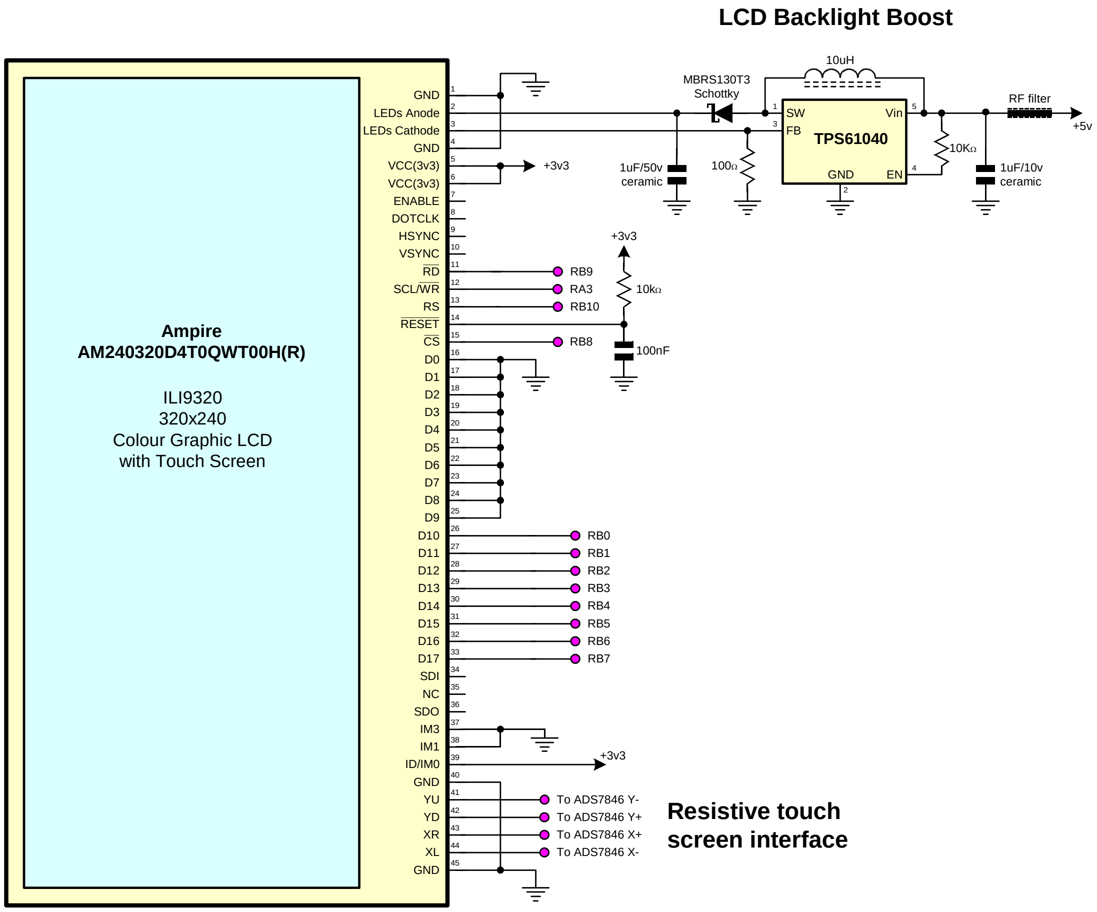


For convenience, there are several colours defined within the ILI9320.inc file. These are:

```
clBlack
clBrightBlue
clBrightGreen
clBrightCyan
clBrightRed
clBrightMagenta
clBrightYellow
clBlue
clGreen
clCyan
clRed
clMagenta
clBrown
clLightGray
clDarkGray
clLightBlue
clLightGreen
clLightCyan
clLightRed
clLightMagenta
clYellow
clWhite
```

More constant values for colours can be added by the user if required.

A suitable circuit for the Ampire AM240320D4T0QWT00H(R) module is shown below:



ILI9320 Colour LCD

ILI9320 colour graphic LCD Declares

There are several declares associated with an ILI9320 colour graphic LCD.

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_CSPin Port **.** Pin

Assigns the Port and Pin that the graphic LCD's CS line will attach to.

There is no default setting for this **Declare** and it must be used within the BASIC program.

Declare **LCD_RSPin** Port **.** Pin
Assigns the Port and Pins that the graphic LCD's RS 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.

Note.
The ILI9320 graphic LCD is a “non-intelligent” type, therefore, a separate character set is required. This is held internally in code memory and is chosen by issuing the **Glcd_SetFont**(pFont) command.

Ptr8, Ptr16, Ptr32, Ptr64

Syntax
Variable **= Ptr8** *(Address)*
Variable **= Ptr16** *(Address)*
Variable **= Ptr32** *(Address)*
Variable **= Ptr64** *(Address)*

or

Ptr8 *(Address)* = *Variable*
Ptr16 *(Address)* = *Variable*
Ptr32 *(Address)* = *Variable*
Ptr64 *(Address)* = *Variable*

Overview
Indirectly address RAM for loading or retrieving using a variable to hold the 16-bit address.

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 retreiving it's RAM location.
- (MyAddress --) Post decrement MyAddress after retreiving it's RAM location.
- (++MyAddress) Pre increment MyAddress before retreiving it's RAM location.
- (--MyAddress) Pre decrement MyAddress before retreiving 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.
Ptr64 will load or retrieve a value with an optional 64-bit post or pre increment or decrement.

8-bit Example.

```
'
' Load and Read 8-bit values indirectly from/to RAM
'
  Device = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600           ' UART1 baud rate
  Declare Hrsout1_Pin = PORTB.14        ' Select pin to be used for TX

  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:
  RPOR7 = 3                         ' Make PPS Pin RP14 U1TX
'
' 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 = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600       ' UART1 baud rate
  Declare Hrsout1_Pin = PORTB.14    ' Select pin is to be used for TX

  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:
  RPOR7 = 3                         ' Make PPS Pin RP14 U1TX
'
' 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 = 24FJ64GA002
  Declare Xtal = 16
  Declare Hserial_Baud = 9600          ' UART1 baud rate
  Declare Hrsout1_Pin = PORTB.14       ' Select pin is to be used for TX

  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:
  RPOR7 = 3                            ' Make PPS Pin RP14 U1TX
'
' 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, cPtr8, cPtr16, cPtr32, cPtr64.**

PulseIn

Syntax
Variable **= PulseIn** *Pin*, *State*

Overview
Change the specified pin to input and measure an input pulse.

Operands
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
```
  Device = 24HJ128GP502
  Declare Xtal = 16

  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 *variabl*e.

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 *Pi*n, *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.

Operands
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 to PORTB Pin5
  Device = 24HJ128GP502
  Declare Xtal = 16

  Low PORTB.5
  PulseOut PORTB.5, 100

' Send a high pulse 1ms long to PORTB Pin5
  PulseOut PORTB.5, 100, High
```

See also : **Counter , PulseIn, RCin.**

Push

Syntax
Push *Variable*, {*Variable*, *Variable* etc}

Overview
Place a single variable or multiple variables onto the microcontroller's stack.

Operands
Variable is a user defined variable of type **Bit**, **Byte**, **Word**, **Dword**, **Float**, **Double**, **Arrays**, **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. The microcontroller's stack is word orientated, therefore all operations are accomplished using 16-bits.

Bit	2 Bytes are pushed that hold the condition of the bit.
Byte	2 Bytes are pushed.
Byte Array	2 Bytes are 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, Mid2 Byte, Mid1 Byte then Low Byte.
Float Array	4 Bytes are pushed. High Byte, Mid2 Byte, Mid1 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.
Double	8 Bytes are pushed. High Byte, Midx Bytes, 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 = 24HJ128GP502
  Declare Xtal = 16
  Declare Stack_Size = 90 ' Increase the stack for holding extra words

  Dim MyWord as Word       ' Create a Word variable
  Dim MyDword as Dword     ' Create a Dword variable

  MyWord = 1234           ' Load the Word variable with a value
  MyDword = 567890         ' Load the Dword variable with a value
  Push MyWord, MyDword     ' Push the Word variable then the Dword variable

  Clear MyWord             ' Clear the Word variable
  Clear MyDword           ' Clear the Dword variable

  Pop MyDword, MyWord      ' Pop the Dword variable then the Word variable
  Print Dec MyWord, "  ", Dec MyDword  ' Display the variables as decimal
```


Example 2

```
' Push a String on to the stack then retrieve it

  Device = 24HJ128GP502
  Declare Xtal = 16
  Declare Stack_Size = 80 ' Increase the stack for holding extra words

  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"
```

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 onto the stack, which requires 2 bytes (remember, the stack is 16-bit orientated).

```
  Push 200
```

All well and good, but what if the recipient popped variable is of a **Dword** or **Float** type.

```
  Pop MyWord
```

Popping from the stack into a **Dword** variable will actually pull 4 bytes from the stack, however, the code above has only pushed two bytes, 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 word to the stack.

```
  Push Byte 12345
```

The **Word** formatter will force any variable or value following it to push only 1 word to the stack:

```
  Push Word 123
```

The **Dword** formatter will force any variable or value following it to push only 2 words to the stack: -

```
  Push Dword 123
```

The **Float** formatter will force any variable or value following it to push only 2 words to the stack, and will convert a constant value into the 2-word floating point format: -

```
Push Float 123.1
```

The **Double** formatter will force any variable or value following it to push only 4 words to the stack, and will convert a constant value into the 4-word 64-bit floating point format: -

```
Push Double 123.1
```

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, 2 words will be placed on the stack because this is a known format.

What is a Stack?

Unlike the 8-bit PIC® microcontrollers, the PIC24® and dsPIC33® devices have a true stack, which is an area of RAM allocated for temporary data storage and call-return address's.

The stack is always present within a PIC24® or dsPIC33® device and is located at the end of the variable RAM. The microcontroller uses it for call and return addresses, but it can also be used for temporary storage of variables. The stack defaults to 60 words, but can be increased or decreased by issuing the **Stack_Size Declare**.

```
Declare Stack_Size = 200
```

The above line of code will increase the stack to 200 words.

Taking the above line of code as an example, we can examine what happens when a variable is pushed on to the 200 word stack, and then popped off again.

Pushing.
When a **Word** variable is pushed onto the stack, the memory map would look like the diagram below: -

```
End Of Stack   Empty RAM
                 ~     ~
                 ~     ~
               Empty RAM
Start Of Stack Contents of the Word Variable
```

Because each element of the stack is 16-bit wide, the contents of the Word variable are placed directly into it. 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: -

```
End Of Stack   Empty RAM
                 ~    ~
                 ~    ~
               Empty RAM
               Contents of the Word Variable
               Contents of the High Word of the Dword Variable
Start Of Stack Contents of the Low Word of the Dword Variable
```

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, and there is a possibility that the microcontroller will cause an exception.

For example, using the above analogy, we need to **Pop** a **Dword** variable first. The **Dword** variable will be popped Low Word first, then the High Word. 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: -

```
End Of Stack   Empty RAM
                 ~    ~
                 ~    ~
               Empty RAM
Start Of Stack Contents of the Word Variable
```

If a **Word** variable was then popped, the stack will be empty, unless it already contains call/return address's. Now what if we popped a **Dword** variable instead of the required **Word** variable? the stack would underflow by one word and definitely cause an exception. The same is true if the stack overflows.

Technical Details of Stack implementation.

The stack implemented by the PIC24® and dsPIC33® microcontroller's 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 underflow or overflow will cause an exception to be triggered.

Whenever a variable is popped from the stack, the stack's memory is not actually cleared, only the stack pointer is moved (WREG15). 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.**

Pwm

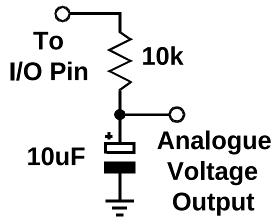
Syntax
Pwm *Pin, Duty*, *Cycles*

Overview
Output pulse-width-modulation on a pin, then return the pin to input state.

Operands
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 very dependant on the oscillator frequency of the microcontroller. The faster the oscillator, the faster the duty cycle.

Notes.
Pwm can be used to generate analogue voltages (0-3.3V) 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, the voltage across it is equal to:-

(duty / 255) * 3.3.

So if *duty* is 100, the capacitor voltage is approximately:

(100 / 255) * 3.3 = 1.29 volts.

See also : **Hpwm, Pulseout, Servo.**

Random

Syntax
Variable **= Random**

or

Random *Variable*

Overview
Generate a pseudo-randomised value.

Operands
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.

Example
```
Device = 24HJ128GP502
Declare Xtal = 16

Var1 = Random                ' Get a random number into Var1
Random Var1                    ' Get a random number into Var1
```

See also: **Seed.**

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.

Operands
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
```
Device = 24HJ128GP502
Declare Xtal = 16
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. 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.

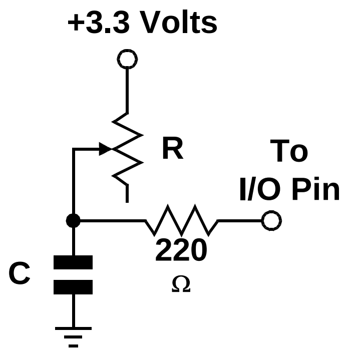


Figure A

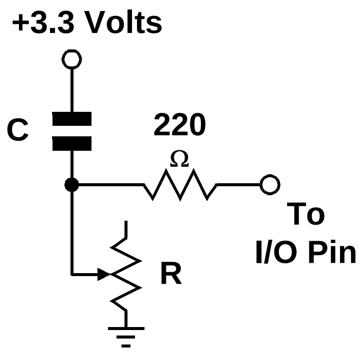


Figure B

The diagrams above show two suitable RC circuits for use with **RCin**. The circuit in figure B is preferred, because of the microcontroller's logic threshold.

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 3.3V. 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 +3.3 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.
```

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
```
Device = 24HJ128GP502
Declare Xtal = 16

Dim MyWord as Word
MyWord = 1
Repeat
  Print Dec MyWord, " "
  DelayMs 200
  Inc MyWord
Until MyWord > 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. `MyWord = MyWord + 1`

Dec. Decrement a variable i.e. `MyWord = MyWord - 1`

The above example shows the equivalent to the **For-Next** loop: -

```
For MyWord = 1 to 10 : Next
```

See also : **While...Wend, For...Next...Step.**

Return

Syntax
Return

Overview
Return from a subroutine.

See also : Call, Gosub.

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 = 24HJ128GP502
  Declare Xtal = 16

  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"
```

Example 2
```
' Copy 5 characters from right of a Quoted Character String to DestString

  Device = 24HJ128GP502
  Declare Xtal = 16

  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"
```

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 = 24HJ128GP502
  Declare Xtal = 16

  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"
```

A third possibility for *Source String* is a Label name, in which case a null terminated Quoted String of Characters is read from code memory.

Example 4

```
' Copy 5 characters from the right of a code memory string into DestString

  Device = 24HJ128GP502
  Declare Xtal = 16

  Dim DestString as String * 20    ' Create a String of 20 characters
' Create a null terminated string of characters in code memory
  Dim Source as Code =  "Hello World", 0

' Copy 5 characters from label Source into the destination string
  DestString = Right$(Source, 5)
  Print DestString               ' Display the result, which will be "World"
```

See also : **Creating and using Strings, Creating and using code memory strings, Len, Left$, Mid$, Str$, ToLower, ToUpper, AddressOf.**

Rol

Syntax
Rol *Variable {,Set or Clear}*

Overview
Bitwise rotate a variable left with or without the microcontroller's Carry flag.

Operands
Variable may be any standard variable type, but not an array.
Set or ***Clear*** are optional parameters that will clear or set the Carry flag before the rotate.
If no parameter is placed after the Variable, the current Carry flag state will be rotated into the LSB (Least Significant Bit) of variable.

Example.
```
' Demonstrate the Rol Command
'
  Device = 24FJ64GA002
  Declare Xtal = 32
  Declare Hserial_Baud = 9600      ' UART1 baud rate
  Declare Hrsout1_Pin = PORTB.14   ' Select the pin for TX with USART1

  Dim Index As Byte
  Dim MyByte As Byte
  Dim Byteout As Byte

  RPOR7 = 3                        ' Make Pin RP14 U1TX
'
' Rotate the carry through MyByte
'
  MyByte = %10000000
  Rol MyByte
  Rol MyByte
  Rol MyByte
  Rol MyByte
  Rol MyByte
  Rol MyByte
  Rol MyByte
  Rol MyByte
'
' Set each bit of MyByte with every rotate
'
  MyByte = %00000000
  For Index  = 0 To 7
    Rol MyByte, Set
    HRSOut Bin8 MyByte, 13
  Next
  HRSOut "----------\r"
'
' Clear each bit of MyByte with every rotate
'
  MyByte = %11111111
  For Index = 0 To 7
    Rol MyByte, Clear
    HRSOut Bin8 MyByte, 13
  Next
  HRSOut "----------\r"
```



```
'
' Transfer the value of MyByte to Byteout, but reversed
'
  MyByte = %10000000
  Byteout = %00000000
  For Index = 0 To 7
    Rol MyByte
    Ror Byteout
    HRSOut Bin8 Byteout, 13
  Next
'
' Configure for internal 8MHz oscillator with PLL
' OSC pins are general purpose I/O
'
  Config Config1 = JTAGEN_OFF, GCP_OFF, GWRP_OFF, BKBUG_OFF,_
                   COE_OFF, ICS_PGx1, FWDTEN_OFF, WINDIS_OFF,_
                   FWPSA_PR128, WDTPOST_PS256
  Config Config2 = IOL1WAY_OFF, COE_OFF, IESO_OFF, FNOSC_FRCPLL,_
                   FCKSM_CSDCMD, OSCIOFNC_ON, POSCMOD_NONE
```

See also: **Ror.**

Ror

Syntax
Ror *Variable {,Set or Clear}*

Overview
Bitwise rotate a variable right with or without the microcontroller's Carry flag.

Operands
Variable may be any standard variable type, but not an array.
Set or ***Clear*** are optional parameters that will clear or set the Carry flag before the rotate.
If no parameter is placed after the *Variable*, the current Carry flag state will be rotated into the MSB (Most Significant Bit) of variable.

Example.
```
' Demonstrate the Ror Command
'
  Device = 24FJ64GA002
  Declare Xtal = 32
  Declare Hserial_Baud = 9600      ' UART1 baud rate
  Declare Hrsout1_Pin = PORTB.14   ' Select the pin for TX with USART1

  Dim Index As Byte
  Dim MyByte As Byte
  Dim Byteout As Byte

  RPOR7 = 3                        ' Make Pin RP14 U1TX
'
' Rotate the carry through MyByte
'
  MyByte = %00000001
  Ror MyByte
  Ror MyByte
  Ror MyByte
  Ror MyByte
  Ror MyByte
  Ror MyByte
  Ror MyByte
  Ror MyByte
'
' Set each bit of MyByte with every rotate
'
  MyByte = %00000000
  For Index  = 0 To 7
    Ror MyByte, Set
    HRSOut Bin8 MyByte, 13
  Next
  HRSOut "----------\r"
'
' Clear each bit of MyByte with every rotate
'
  MyByte = %11111111
  For Index = 0 To 7
    Ror MyByte, Clear
    HRSOut Bin8 MyByte, 13
  Next
  HRSOut "----------\r"
```



```
'
' Transfer the value of MyByte to Byteout, but reversed
'
  MyByte = %00000001
  Byteout = %00000000
  For Index = 0 To 7
    Ror MyByte
    Rol Byteout
    HRSOut Bin8 Byteout, 13
  Next
'
' Configure for internal 8MHz oscillator with PLL
' OSC pins are general purpose I/O
'
  Config Config1 = JTAGEN_OFF, GCP_OFF, GWRP_OFF, BKBUG_OFF,_
                   COE_OFF, ICS_PGx1, FWDTEN_OFF, WINDIS_OFF,_
                   FWPSA_PR128, WDTPOST_PS256
  Config Config2 = IOL1WAY_OFF, COE_OFF, IESO_OFF, FNOSC_FRCPLL,_
                   FCKSM_CSDCMD, OSCIOFNC_ON, POSCMOD_NONE
```

See also: **Rol.**

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.

Operands
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
```
  Device = 24HJ128GP502
  Declare Xtal = 16
  Declare Rsin_Timeout = 2000       ' Timeout after 2 seconds
  Dim MyByte as Byte
  Dim MyWord as Word
  MyByte = Rsin, {Label}
  Rsin MyByte, MyWord
  Rsin { Label }, MyByte, MyWord

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 microcontroller.

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 microcontroller was 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 is 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 foolproof. 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 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 **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 microcontroller 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 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 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 microcontroller, 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 **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 device will receive the data correctly.

Notes.
Rsin is oscillator independent as long as the crystal frequency is declared at the top of the program.

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.

Operands
Item may be a constant, variable, expression, or string list.
There are no operands as such, instead there are *modifiers*. For example, if an at sign'@' precedes an *Ite*m, the ASCII representation for each digit is transmitted.

The modifiers are listed below: -

Modifier	**Operation**
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 Label	Send string data defined in code memory.

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 MyFloat as Float
MyFloat = 3.145
Rsout Dec2 MyFloat     ' Send 2 values after the decimal point
```

The above program will transmit the ASCII representation of the value 3.14

If the digit after the **Dec** modifier is omitted, then 3 values will be displayed after the decimal point.


```
Dim MyFloat as Float
MyFloat = 3.1456
Rsout Dec MyFloat   ' 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 MyFloat as Float
MyFloat = -3.1456
Rsout Dec MyFloat   ' Send 3 values after the decimal point
```

The above program will transmit the ASCII representation of the value -3.145

Example

```
Device = 24HJ128GP502
Declare Xtal = 16
Dim Var1 as Byte
Dim MyWord as Word
Dim MyDword 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 "MyDword= ", Hex6 MyDword  ' Display 6 hex chars of a Dword variable
```

Example 3 will produce the text "$-1234" on a serial terminal.

The **Cstr** modifier is used in conjunction with code memory strings. The **Dim as Code** directive is used for initially creating the string of characters: -

```
Dim String1 as Code = "Hello World", 0
```

The above line of case 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 code memory 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 = 24HJ128GP502
Declare Xtal = 16
Rsout "Hello World\r"
Rsout "How are you?\r"
Rsout "I am fine!\r"
```

Now using the **Cstr** modifier: -

```
Dim Text1 as Code = "Hello World\r", 0
Dim Text2 as Code = "How are you?\r", 0
Dim Text3 as Code = "I am fine!\r", 0

Rsout Cstr Text1
Rsout Cstr Text2
Rsout Cstr Text3
```

Again, note the null terminators after the ASCII text in the code memory data. Without these, the microcontroller will continue to transmit data until a value of 0 is reached.

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 microcontroller 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 device.

Declare Rsout_Mode Inverted, True 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 and above.

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.

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.

Operands
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 = 24HJ128GP502
  Declare Xtal = 16

  Dim MyWord as Word

  Seed $0345
  Cls
  While
    MyWord = Random
    Print At 1,1,Dec MyWord, "     "
    DelayMs 500
  Wend
```

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.

Operands
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
```
' Result will return a value of 255 if no valid condition was met
  Device = 24HJ128GP502
  Declare Xtal = 16
  Dim MyByte as Byte
  Dim Result as Byte
  DelayMs 100           ' Wait for things to stabilise
  Cls                   ' Clear the LCD
  Result = 0            ' Clear the result variable before we start
  MyByte = 1            ' Variable to base the conditions upon
  Select MyByte
    Case 1              ' Is MyByte equal to 1?
      Result = 1        ' Load Result with 1 if yes
    Case 2              ' Is MyByte equal to 2?
      Result = 2        ' Load Result with 2 if yes
    Case 3              ' Is MyByte 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
```


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 operands <>, <, >, >= 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.**

Servo

Syntax
Servo *Pin*, *Rotation Value*

Overview
Control a remote control type servo motor.

Operands
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 PORTB

  Device = 24HJ128GP502
  Declare Xtal = 16

  Dim Pos as Word            ' Servo Position
  Symbol Pin = PORTB.3       ' Alias the servo pin

  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
  While
    If PORTA.0 = 0 And Pos < 3000 Then Inc Pos    ' Move servo left
    If PORTA.1 = 0 Then Pos = 1500                ' Centre servo
    If PORTA.2 = 0 And Pos > 0 Then Dec Pos      ' Move servo right
    Servo Pin, Pos
    DelayMs 5                                     ' Servo update rate
    Hrsout "Position=", Dec Pos, 13
  Wend
```

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 Proton24 is extremely easy. The **Servo** command generates a pulse in 1 microsecond (µs) units, so the following code would command a servo to its centred position and hold it there: -

```
While
  Servo PORTA.0, 1500
  DelayMs 20
Wend
```

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.

Operands
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 = 24HJ128GP502
  Declare Xtal = 16
  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 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. 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

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.

Operands
Variable can be any variable or register.
Variable*.*Bit can be any variable and bit combination.

Example
```
Set MyVar.3   ' Set bit 3 of MyVar
Set MyByte    ' Load MyByte with the value of 255
Set SR.0      ' Set the Carry flag high
Set Array     ' Set all of an Array variable.
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.

Operands
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 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 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 Shift in 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: -

```
MyWord = Shin DTA, CLK, MsbPre, 16
```


Shout

Syntax
Shout *Dpin*, *Cpin*, *Mode*, **[***OutputData* {*Bits*} {,*OutputData* {*Bits*}..} **]**

Overview
Shift data out to a synchronous serial device.

Operands
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 *Dpi*n) and will generate a clock signal on I/O CLK (the *Cpi*n). The **Shout** command will generate eight clock pulses while writing each bit (of the 8-bit value 250) onto the data pin *(Dpi*n). 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.

Sleep

Syntax
Sleep { *Length* }

Overview
Places the microcontroller into low power mode for approx *n* seconds.

Operators
Length is an optional variable or constant (1-16383) that specifies the duration of sleep in approximate 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 an internal or external influence wakes it up.

Example
```
Symbol MyLED = PORTA.0
While
  High MyLED          ' Turn LED on.
  DelayMs 1000        ' Wait 1 second.
  Low MyLED           ' Turn LED off.
  Sleep 60            ' Sleep for 1 minute.
Wend
```

Notes.
Sleep will place the device into a low power mode for the specified period of seconds. Period is 14-bits, so delays of up to 16383 seconds are the limit. **Sleep** uses the Watchdog Timer so it is independent of the oscillator frequency.

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.

The watchdog must be enabled an set to a postscaler value of 1:256 for sleep to work correctly.

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.

Operands
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 = 24HJ128GP502
  Declare Xtal = 16

  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 PIC24® and dsPIC33®, 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.**

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.

Operands
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

  Device = 24HJ128GP502
  Declare Xtal = 16
  Dim String1[21] as Byte    ' Create a Byte array with 21 elements

  DelayMs 100          ' Wait for things to stabilise
  Cls                  ' Clear the LCD
  Strn String1 = "Hello World"
  ' Load String1 with characters and null terminate it
  Print Str String1    ' Display the string
```

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.

Operands

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.
  Device = 24HJ128GP502
  Declare Xtal = 16
'
' Create a byte array to hold converted value, and null terminator
  Dim MyString as String * 12
  Dim MyWord1 as Word

  DelayMs 100              ' Wait for things to stabilise
  Cls                      ' Clear the LCD
  MyWord1 = 1234           ' Load the variable with a value
  Strn MyString = Str$(Dec MyWord1)  ' Convert the Integer to a String
  Print MyString           ' Display the string
```

Example 2

```
' Convert a Dword variable to a String of characters in a Byte array.
  Device = 24HJ128GP502
  Declare Xtal = 16

  Dim MyString as String * 12
  Dim MyDword1 as Dword
  DelayMs 100                ' Wait for things to stabilise
  Cls                        ' Clear the LCD
  MyDword1 = 1234            ' Load the variable with a value
  Strn MyString = Str$(Dec MyDword1) ' Convert the Integer to a String
  Print MyString             ' Display the string
```

Example 3

```
' Convert a Float variable to a String of characters in a Byte array.
  Device = 24HJ128GP502
  Declare Xtal = 16

  Dim MyString as String * 12
  Dim MyFloat1 as Float
  DelayMs 100                ' Wait for things to stabilise
  Cls                        ' Clear the LCD
  MyFloat1 = 3.14            ' Load the variable with a value
  Strn MyString = Str$(Dec MyFloat1) ' Convert the Float to a String
  Print MyString             ' Display the string
```

Example 4

```
' Convert a Word variable to a Binary String of characters in an array.
  Device = 24HJ128GP502
  Declare Xtal = 16

  Dim MyString as String * 32
  Dim MyWord1 as Word
  DelayMs 100                ' Wait for things to stabilise
  Cls                        ' Clear the LCD
  MyWord1 = 1234             ' Load the variable with a value
  Strn MyString = Str$(Bin MyWord1)  ' Convert the Integer to a String
  Print MyString             ' Display the string
```

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.

Operands
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

Operands
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 = 3.0     ' Create a floating point constant with the value 3
```

Floating point constant can also be created using expressions.

```
' Create a floating point constant holding the result of the expression
Symbol Quanta = 3.3 / 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 *Variable {.Bit}*

Overview
Reverses a variable or pin. If a Port's pin is chosen as the operand, it will first be ser to output mode. i.e. Changing 0 to 1 and 1 to 0.

Operands
Variable {.Bit} can be any valid variable, variable and bit, or Port and Bit combination.

Example
```
High PORTB.0       ' Set bit 0 of PORTB high
Toggle PORTB.0     ' And now reverse the bit

Toggle Var1.0      ' Reverse bit-0 of Var1
Toggle Var1        ' Reverse the whole of Var1
```

See also : **High, Low.**

ToLower

Syntax
Destination String = **ToLower (***Source String***)**

Overview
Convert the characters from a source string to lower case.

Operands
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 code memory.

Example 1
```
' Convert the characters from SourceString to lowercase into DestString

  Device = 24HJ128GP502
  Declare Xtal = 16
  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"
```

Example 2
```
' Convert the characters from a Quoted Character String to lowercase
' into DestString

  Device = 24HJ128GP502
  Declare Xtal = 16
  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"
```

Example 3
```
' Convert to lowercase from SourceString into DestString using a pointer to
' SourceString

  Device = 24HJ128GP502
  Declare Xtal = 16
  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"
```


Example 4

```
' Convert chars from a code memory string to lowercase
' and place into DestString

  Device = 24HJ128GP502
  Declare Xtal = 16
  Dim DestString as String * 20    ' Create a String of 20 characters
' Create a null terminated string of characters in code memory
  Dim Source as Code = "HELLO WORLD", 0

  DestString = ToLower(Source)   ' Convert to lowercase
  Print DestString          ' Display the result, which will be "hello world"
```

See also : **Creating and using Strings, Creating and using code memory strings, 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.

Operands
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 code memory.

Example 1
```
' Convert the characters from SourceString to UpperCase and place into
' DestString

  Device = 24HJ128GP502
  Declare Xtal = 16
  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"
```

Example 2
```
' Convert the chars from a Quoted Character String to UpperCase
' and place into DestString

  Device = 24HJ128GP502
  Declare Xtal = 16
  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"
```

Example 3
```
' Convert to UpperCase from SourceString into DestString using a pointer to
' SourceString

  Device = 24HJ128GP502
  Declare Xtal = 16
  Dim SourceString as String * 20  ' Create a String of 20 characters
  Dim DestString as String * 20    ' Create another String
  Dim StringAddr as Word  ' Create a Word variable to hold address

  SourceString = "hello world"  ' Load the source string with characters
  ' 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"
```

Example 4

```
' Convert chars from a code memory string to uppercase
' and place into DestString

  Device = 24HJ128GP502
  Declare Xtal = 16
  Dim DestString as String * 20  ' Create a String of 20 characters
' Create a null terminated string of characters in code memory
  Dim Source as Code = hello world", 0

  DestString = ToUpper(Source) ' Convert to uppercase
  Print DestString    ' Display the result, which will be "HELLO WORLD"
```

See also : **Creating and using Strings, Creating and using code memory strings, Len, Left$, Mid$, Right$, Str$, ToLower, AddressOf .**

Touch_Active

Syntax
Var = **Touch_Active**

Overview
Indicates if the graphic LCD's resistive touch membrane has been touched.

Assignment
Var can be any valid variable type and holds 1 if the touch screen membrane has been touched with sufficient force.

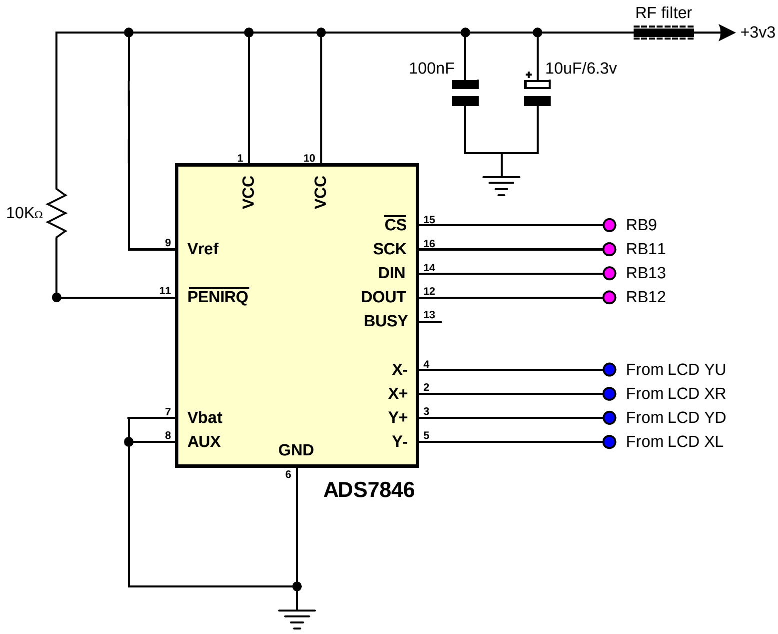
Example
```
' Demonstrate the Touch_Active command
'
  Device = 24HJ128GP502
  Declare Xtal = 79.23
  Declare Hserial_Baud = 9600    ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14 ' Select the pin used for TX with USART1
'
' Setup the ADS7846 touchscreen chip's pins
'
  Declare Touch_DINPin = PORTB.13  ' Connect to the ADS7846 DIN pin
  Declare Touch_DOUTPin = PORTB.12 ' Connect to the ADS7846 DOUT pin
  Declare Touch_CLKPin = PORTB.11  ' Connect to the ADS7846 CLK pin
  Declare Touch_CSPin = PORTB.9    ' Connect to the ADS7846 CS pin

  Include "TouchScreen.inc" ' Load the touchscreen routines into the program

'-------------------------------------------------------------------------
Main:
' Configure the internal oscillator to operate the device at 79.23MHz
'
  PLL_Setup(43, 2, 2, $0300)
  RPOR7 = 3                        ' Make Pin RP14 U1TX

  While
    If Touch_Active = 1 Then       ' Has the LCD been touched?
      HRSOut "LCD is Touched\r"    ' Yes. So transmit a message serially
      DelayMS 200
    EndIf
  Wend
'-------------------------------------------------------------------------
' Configure for internal 7.37MHz oscillator with PLL
' OSC pins are general purpose I/O
'
  Config FBS = BWRP_WRPROTECT_OFF, BSS_NO_FLASH, BSS_NO_BOOT_CODE
  Config FSS = SWRP_WRPROTECT_OFF, SSS_NO_FLASH, RSS_NO_SEC_RAM
  Config FGS = GWRP_OFF, GCP_OFF
  Config FOSCSEL = FNOSC_FRCPLL, IESO_ON
  Config FOSC = POSCMD_NONE, OSCIOFNC_ON, IOL1WAY_OFF, FCKSM_CSDCMD
  Config FWDT = WDTPOST_PS256, WINDIS_OFF, FWDTEN_OFF
  Config FPOR = FPWRT_PWR128, ALTI2C_OFF
  Config FICD = ICS_PGD1, JTAGEN_OFF
```

Notes.

The touch screen commands used by the compiler are for use with an ADS7846 touch screen controller device. This device uses an SPI interface and connects to a 4-wire resistive touch screen membrane to give X and Y coordinates, as well as touch pressure.

The routines must be incorporated into the BASIC program by use of an include file named ***"TouchScreen.inc".*** This is written in Proton24 BASIC so that modifications or improvements are easy. It also exposes how the touch screen is interfaced with.

A suitable circuit for the ADS7846 touch screen controller is shown below:



ADS7846 Touch controller circuit

See Also. Touch_Read, Touch_HotSpot

Touch_Read

Syntax
Var = **Touch_Read**

Overview
Get the X and Y pixel coordinates from the graphic LCD's resistive touch membrane.

Assignment
Var can be any valid variable type and holds 1 if the touch screen membrane has been touched within its bounds.
Two variables are loaded with the X and Y pixel coordinates. These are:
Touch_Xpos holds the X position of the touch (0 to 239)
Touch_Ypos holds the Y position of the touch (0 to 319)

Example
```
' Demonstrate the Touch_Read command
'
  Device = 24HJ128GP502
  Declare Xtal = 79.23
'
' Setup the ADS7846 touchscreen chip's pins
'
  Declare Touch_DINPin = PORTB.13       ' Connect to the ADS7846 DIN pin
  Declare Touch_DOUTPin = PORTB.12      ' Connect to the ADS7846 DOUT pin
  Declare Touch_CLKPin = PORTB.11       ' Connect to the ADS7846 CLK pin
  Declare Touch_CSPin = PORTB.9         ' Connect to the ADS7846 CS pin

  Include "TouchScreen.inc" ' Load the touchscreen routines into the program

' Configure the internal oscillator to operate the device at 79.23MHz
'
  PLL_Setup(43, 2, 2, $0300)
  While
    If Touch_Active = 1 Then          ' Has the LCD been touched?
      If Touch_Read = 1 Then          ' Is the touch within bounds?
        HRSOut "X Touch = ", Dec Touch_Xpos, 13
        HRSOut "Y Touch = ", Dec Touch_Ypos, 13
        DelayMS 200
      EndIf
    EndIf
  Wend
'-------------------------------------------------------------------------
' Configure for internal 7.37MHz oscillator with PLL
' OSC pins are general purpose I/O
'
  Config FBS = BWRP_WRPROTECT_OFF, BSS_NO_FLASH, BSS_NO_BOOT_CODE
  Config FSS = SWRP_WRPROTECT_OFF, SSS_NO_FLASH, RSS_NO_SEC_RAM
  Config FGS = GWRP_OFF, GCP_OFF
  Config FOSCSEL = FNOSC_FRCPLL, IESO_ON
  Config FOSC = POSCMD_NONE, OSCIOFNC_ON, IOL1WAY_OFF, FCKSM_CSDCMD
  Config FWDT = WDTPOST_PS256, WINDIS_OFF, FWDTEN_OFF
  Config FPOR = FPWRT_PWR128, ALTI2C_OFF
  Config FICD = ICS_PGD1, JTAGEN_OFF
```


Notes.
The touch screen commands used by the compiler are for use with an ADS7846 touch screen controller device. This device uses an SPI interface and connects to a 4-wire resistive touch screen membrane to give X and Y coordinates, as well as touch pressure.

The routines must be incorporated into the BASIC program by use of an include file named ***"TouchScreen.inc".*** This is written in Proton24 BASIC so that modifications or improvements are easy. It also exposes how the touch screen is interfaced with.

See Also. Touch_Active, Touch_HotSpot

Touch_HotSpot

Syntax
Var = **Touch_HotSpot** *Xpos Start*, *Ypos Start*, *Xpos End*, *Ypos End*

Overview
Indicate when a user defined area on the graphic LCD's resistive touch membrane has been touched.

Operands
Var can be any valid variable type and holds 1 if the touch screen membrane has been touched within the window's bounds.
Xpos Start can be any valid variable type that holds the X position for the start of the touch window. Can be a value from 0 to the LCD's X resolution.
Ypos Start can be any valid variable type that holds the Y position for the start of the touch window. Can be a value from 0 to the LCD's Y resolution.
Xpos End can be any valid variable type that holds the X position for the end of the touch window. Can be a value from 0 to the LCD's X resolution.
Ypos End can be any valid variable type that holds the Y position for the end of the touch window. Can be a value from 0 to the LCD's Y resolution.

The Windowed area's X and Y start positions are top left of the LCD, as in the other pixel based routines.

Example
```
' Demonstrate the Touch_HotSpot command
'
  Device = 24HJ128GP502
  Declare Xtal = 79.23
  Declare Hserial_Baud = 9600      ' USART1 baud rate
  Declare Hrsout1_Pin = PORTB.14   ' Select the pin used for TX with USART1
'
' Setup the touchscreen chip's pins
'
  Declare Touch_DINPin = PORTB.13    ' Connect to the ADS7846 DIN pin
  Declare Touch_DOUTPin = PORTB.12   ' Connect to the ADS7846 DOUT pin
  Declare Touch_CLKPin = PORTB.11    ' Connect to the ADS7846 CLK pin
  Declare Touch_CSPin = PORTB.9      ' Connect to the ADS7846 CS pin

  Include "TouchScreen.inc" ' Load the touchscreen routines into the program

' Configure the internal oscillator to operate the device at 79.23MHz
'
  PLL_Setup(43, 2, 2, $0300)
  RPOR7 = 3                        ' Make PPS Pin RP14 U1TX
'
' Transmit a message if the LCD is touched within a window 40 pixels square
  While
    If Touch_Active = 1 Then       ' Has the LCD been touched?
      Touch_Read                   ' Read the touch X and Y
      If Touch_HotSpot 0, 0, 40, 40 = 1 Then
        HRSOut "Touched at X ", Dec Touch_Xpos, ", Y ", Dec Touch_Ypos, 13
        DelayMS 100
      EndIf
    EndIf
  Wend
```

```
'-------------------------------------------------------------------------
' Configure for internal 7.37MHz oscillator with PLL
' OSC pins are general purpose I/O
'
  Config FBS = BWRP_WRPROTECT_OFF, BSS_NO_FLASH, BSS_NO_BOOT_CODE
  Config FSS = SWRP_WRPROTECT_OFF, SSS_NO_FLASH, RSS_NO_SEC_RAM
  Config FGS = GWRP_OFF, GCP_OFF
  Config FOSCSEL = FNOSC_FRCPLL, IESO_ON
  Config FOSC = POSCMD_NONE, OSCIOFNC_ON, IOL1WAY_OFF, FCKSM_CSDCMD
  Config FWDT = WDTPOST_PS256, WINDIS_OFF, FWDTEN_OFF
  Config FPOR = FPWRT_PWR128, ALTI2C_OFF
  Config FICD = ICS_PGD1, JTAGEN_OFF
```

Notes.
The touch screen commands used by the compiler are for use with an ADS7846 touch screen controller device. This device uses an SPI interface and connects to a 4-wire resistive touch screen membrane to give X and Y coordinates, as well as touch pressure.

The routines must be incorporated into the BASIC program by use of an include file named ***"TouchScreen.inc".*** This is written in Proton24 BASIC so that modifications or improvements are easy. It also exposes how the touch screen is interfaced with.

See Also. Touch_Active, Touch_Read.

Toshiba_Command

Syntax
Toshiba_Command *Command, Value*

Overview
Send a command with or without parameters to a Toshiba T6963 graphic LCD.

Operands
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 T6963C datasheet.

Example
```
'
' Toshiba T6963C Command demo
'
  Device = 24FJ64GA002
  Declare Xtal = 16
'
' Toshiba T6963C graphic LCD Pin configuration
'
  Declare LCD_Type = Toshiba          ' LCD's type is Toshiba T6963C
  Declare LCD_DTPort = PORTB.Byte0    ' The LCD's 8-bit Data port
  Declare LCD_WRPin = PORTB.12        ' The LCD's WR pin
  Declare LCD_RDPin = PORTB.11        ' The LCD's RD pin
  Declare LCD_CEPin = PORTB.10        ' The LCD's CE pin
  Declare LCD_CDPin = PORTB.8         ' The LCD's CD pin
  Declare LCD_RSTPin = PORTB.9        ' The LCD's RST pin (optional)
'
' Toshiba T6963C graphic LCD setup configuration
'
  Declare LCD_Font_Width = 8          ' The font width ( 6 or 8)
  Declare LCD_X_Res = 128             ' The X resolution of the LCD
  Declare LCD_Y_Res = 64              ' The Y resolution of the LCD
  Declare LCD_Text_Home_Address = 0   ' The home address of the LCD
  Declare LCD_RAM_Size = 8192         ' The amount of RAM the LCD contains
  Declare LCD_Text_Pages = 1          ' The amount of text pages required

  Include "T6963C.inc"  ' Load the Toshiba T6963C routines into the program
```



```
    Dim PanLoop As Byte
    Dim Ypos As Byte
'----------------------------------------------------------------------------
Main:
  Cls                                            ' Clear Text and Graphic RAM
'
' 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
    '
    ' Increment the Text home address
    '
    For PanLoop = 0 To 23
      Toshiba_Command cT6963_SET_TEXT_HOME_ADDRESS , Word PanLoop
      DelayMS 200
    Next
    DelayMS 200
    '
    ' Decrement the Text home address
    '
    For PanLoop = 23 To 0 Step -1
      Toshiba_Command cT6963_SET_TEXT_HOME_ADDRESS , Word PanLoop
      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_WRPort The Port where the LCD's WR pin is attached.
__LCD_RDPort The Port where the LCD's RD pin is attached.
__LCD_CEPort The Port where the LCD's CE pin is attached.
__LCD_CDPort The Port where the LCD's CD pin is attached.
__LCD_RSTPort The Port where the LCD's RST pin is attached.

Constants.

__LCD_Type	The type of LCD targeted. 0 = Alphanumeric, 1 = Samsung, 2 = Toshiba.
__LCD_WRPin	The Pin where the LCD's WR line is attached.
__LCD_RDPin	The Pin where the LCD's RD line is attached.
__LCD_CEPin	The Pin where the LCD's CE line is attached.
__LCD_CDPin	The Pin where the LCD's CD line is attached.
__LCD_RSTPin	The Pin where the LCD's RST line is attached.
__LCD_Text_Pages	The amount of TEXT pages chosen.
__LCD_Graphic_Pages	The amount of Graphic pages chosen.
__LCD_RAM_Size	The amount of RAM that the LCD contains.
__LCD_X_Res	The X resolution of the LCD. i.e. Horizontal pixels.
__LCD_Y_Res	The Y resolution of the LCD. i.e. Vertical pixels.
__LCD_Font_Width	The width of the font. i.e. 6 or 8.
__LCD_Text_AREA	The amount of characters on a single line of TEXT RAM.
__LCD_Graphic_AREA	The amount of characters on a single line of Graphic RAM.
__LCD_Text_Home_Address	The Starting address of the TEXT RAM.
__LCD_Graphic_Home_Address	The Starting address of the Graphic RAM.
__LCD_CGRAM_Home_Address	The Starting address of the CG RAM.
__LCD_End_OF_Graphic_RAM	The Ending address of Graphic RAM.
__LCD_CGRAM_OFFset	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.

Toshiba_UDG

Syntax
Toshiba_UDG *Character*, **[***Value {, Values }* **]**

Overview
Create **U**ser **D**efined **G**raphics for a Toshiba T6963 graphic LCD.

Operands
Character can be a constant, variable, or expression, that contains the character to define. User defined characters start from 160 to 255.
Value\s 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
```
'
' Toshiba T6963C UDG (User Defined Graphics) demo
'
  Device = 24FJ64GA002
  Declare Xtal = 16
'
' Toshiba T6963C graphic LCD Pin configuration
'
  Declare LCD_Type = Toshiba            ' LCD's type is Toshiba T6963C
  Declare LCD_DTPort = PORTB.Byte0      ' The LCD's 8-bit Data port
  Declare LCD_WRPin = PORTB.12          ' The LCD's WR pin
  Declare LCD_RDPin = PORTB.11          ' The LCD's RD pin
  Declare LCD_CEPin = PORTB.10          ' The LCD's CE pin
  Declare LCD_CDPin = PORTB.8           ' The LCD's CD pin
  Declare LCD_RSTPin = PORTB.9          ' The LCD's RST pin (optional)
'
' Toshiba T6963C graphic LCD setup configuration
'
  Declare LCD_Font_Width = 8            ' The font width ( 6 or 8)
  Declare LCD_X_Res = 128               ' The X resolution of the LCD
  Declare LCD_Y_Res = 64                ' The Y resolution of the LCD
  Declare LCD_Text_Home_Address = 0     ' The home address of the LCD
  Declare LCD_RAM_Size = 8192           ' The amount of RAM the LCD contains
  Declare LCD_Text_Pages = 2            ' The amount of text pages required

  Include "T6963C.inc"    ' Load the Toshiba T6963C routines into the pro-
gram

'---------------------------------------------------------------------------
Main:
  Dim UDG_Array[10] As Byte = $18, $18, $99, $DB, $7E, $3C, $18, $18
  Dim UDG_Code As Code = $30, $18, $0C, $FF, $FF, $0C, $18, $30

  Cls Text                                          ' Clear Text RAM
'
' Print the user defined graphic characters 160, 161 and 162 on the LCD
'
  Print At 1,0,"Char 160 = ",160,_
        At 2,0,"Char 161 = ",161,_
        At 3,0,"Char 162 = ",162
```

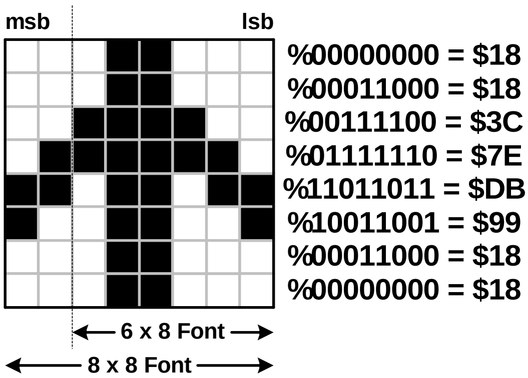
```
'
' Create the UDG (User Defined Graphics) for three characters
'
  Toshiba_Udg 160,[UDG_Code\8]
  Toshiba_Udg 161,[Str UDG_Array\8]
  Toshiba_Udg 162,[$0C, $18, $30, $FF, $FF, $30, $18, $0C]
```

Notes.
User Defined Graphic values can be stored in code memory, and retrieved by the use of a label name associated with the **Dim as Code** table:

```
  Dim UDG_2 as Code = $30, $18, $0C, $FF, $FF, $0C, $18, $30

  Toshiba_UDG 161, [UDG_2\8]
```

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.

UnPlot

Syntax
UnPlot *Ypos, Xpos*

Overview
Clear an individual pixel on a graphic LCD.

Operands
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 = 24HJ128GP502
  Declare Xtal = 16
'
' KS0108 graphic LCD declares
'
  Declare LCD_Type = Samsung        ' Setup for a Samsung KS0108 graphic LCD
  Declare LCD_DTPort = PORTB.Byte0
  Declare LCD_CS1Pin = PORTB.8
  Declare LCD_CS2Pin = PORTB.9
  Declare LCD_ENPin = PORTB.10
  Declare LCD_RSPin = PORTB.11
  Declare LCD_RWPin = PORTB.12

  Dim Xpos as Byte

  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.**

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.

Operands
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
  Device = 24HJ128GP502
  Declare Xtal = 16
  Dim String1 as String * 10   ' Create a String
  Dim MyWord as Word           ' Create a variable to hold result
  DelayMs 100                  ' Wait for things to stabilise
  Cls                          ' Clear the LCD
  String1 = "12AF"             ' Load the String with Hex ASCII
  MyWord = Val(String1,Hex)    ' Convert the String into an integer
  Print Hex MyWord             ' Display the integer as Hex
```

Example 2
```
' Convert a string of decimal characters to an integer
  Device = 24HJ128GP502
  Declare Xtal = 16
  Dim String1 as String * 10   ' Create a String
  Dim MyWord as Word           ' Create a variable to hold result
  DelayMs 100                  ' Wait for things to stabilise
  Cls                          ' Clear the LCD
  String1 = "1234"             ' Load the String with Decimal ASCII
  MyWord = Val(String1,Dec)    ' Convert the String into an integer
  Print Dec MyWord             ' Display the integer as Decimal
```

Example 3
```
' Convert a string of binary characters to an integer
  Device = 24HJ128GP502
  Declare Xtal = 16
  Dim String1 as String * 17   ' Create a String
  Dim MyWord as Word           ' Create a variable to hold result
  DelayMs 100                  ' Wait for things to stabilise
  Cls                          ' Clear the LCD
  String1 = "1010101010000000" ' Load the String with Binary ASCII
  MyWord = Val(String1,Bin)    ' Convert the String into an integer
  Print Bin MyWord             ' Display the integer as Binary
```

Notes.
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.

```
 Device = 24HJ128GP502
 Declare Xtal = 16

Dim String1 as String * 10        ' Create a String
DelayMs 100                       ' Wait for things to stabilise
Cls                               ' Clear the LCD
String1 = "123"                   ' Load the String with Decimal ASCII
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
```

See also: **Str, Strn, Str$.**

AddressOf

Syntax
Assignment Variable = **AddressOf (** *Variable or Label* ***)***

Overview
Returns the address of a variable in RAM, or a label in code memory. Commonly known as a pointer.

Operands
Assignment Variable can be any of the compiler's variable types, and will receive the *variable's* or *label's* address.
Variable or Label can be any variable type used in the BASIC program, or it can be a label name, in which case, it will return the code memory address.

While...Wend

Syntax
While *Condition*
Instructions
Instructions
Wend

or

While *Condition* { *Instructions* } **: Wend**

or

While
Instructions
Instructions
Wend

Overview
Execute a block of instructions while a condition is true, unless no condition is placed after **While**, in which case and infinite loop will be created.

Example
```
MyVar = 1
While MyVar <= 10
  Print Dec MyVar, " "
  MyVar = MyVar + 1
Wend
```

or

```
While PORTA.0 = 1 : Wend   ' Wait for a change on the Port
```

or

```
MyVar = 1
While
  Print at 1,1, Dec MyVar, " "
  MyVar = MyVar + 1
Wend
```

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. If not condition is placed after **While**, an infinite loop will be created. A no condition **While-Wend** is only valid when both are on a separate line.

See also : **If-Then, Repeat-Until, For-Next.**

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, pre-processes 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 pseudo 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 throughout 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 Proton24 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 operands 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.

$ifndef serves for the exact opposite of **$ifdef**. 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 Unique Name
  $define Unique Name
  { 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 **$define** *Unique Name* will not have been created, thus allowing the inclusion of the code between **$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.

Unique Name 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 = _24FJ64GA002 or _device = _24FJ128GA002 and _core = 24
  { BASIC Code Here }
$endif
```

There are several built in user defines that will help separate blocks of code. These are:-

- **_device**. This holds the device name, as a string. i.e. _24FJ64GA002 etc.
- **_type**. This hold the type of PIC24. E, F or H or dsPIC33. F or E:
  For PIC24E, **_type** will hold the ASCII string _PIC24E
  For PIC24F, **_type** will hold the ASCII string _PIC24F
  For PIC24H, **_type** will hold the ASCII string _PIC24H
  For dsPIC33E, **_type** will hold the ASCII string _DSPIC33E
  For dsPIC33F, **_type** will hold the ASCII string _DSPIC33F
- **_core**.This holds the device's core. i.e. 24 or 33
- **_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 *bytes*.
- **_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.
- **_usart**. This holds the amount of USARTS the device has. i.e. 0, 1, 2, 3, or 4

The values for the user defines are taken from the compiler's .def files, and are only available if the compiler's **Device** directive is included within the BASIC program.

Also within the compiler's .def files are all the device's SFRs (Special Function Registers) and SFR bit names. The SFR names are preceded by an underscore so they do not clash with the assembler's SFR names. For example:

WREG0 is _WREG0
WREG12 is _WREG12

The SFR names are useful for compiling a piece of code only if that particular SFR is present in the device being used:

```
$ifdef  _T1CON
  { BASIC Code Here }
$endif
```

The SFR bit names are extremely useful within the BASIC program because they circumvent any differences in the device's makeup. For example, in order to access a devices Carry flag, use: SRbits_C

All the bitnames follow the same rule, where the SFR name is first, followed by the text "bits_", followed by the bit name. Below are a few examples:

T1CONbits_TCS
T1CONbits_TSYNC
T1CONbits_TGATE
T1CONbits_TSIDL
T1CONbits_TON
T1CONbits_TCKPS0
T1CONbits_TCKPS1

$else
This toggles the logical value of the current conditional block. What follows is evaluated if the preceding condition was not met.

$endif
This ends a conditional block started by the **$if**, **$ifdef** or **$ifndef** directives.

$elseif *expression*
This directive can be used to avoid nested **$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 behaviour 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
```


Protected Proton24 Compiler Words

Below is a list of protected words that the compiler, assembler or linker 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, AddressOf, ADC_Resolution, Adcin, Adin, Adin_Delay, Adin_Res, Adin_Stime, Adin_Tad, Asin, Asm, Atan, Atan2, Available_RAM
(B)
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, Bstart, Bstop, 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
(C)
Call, Case, Cblock, CCP1_Pin, CCP2_Pin, CCP3_Pin, CCP4_Pin, CCP5_Pin, Cdata, Cerase, Chr$, Circle, Clear, ClearBit, Cls, Code, Config, Constant, Continue, Core, Cos, Count, Counter, CPtr8, CPtr16, CPtr32, CPtr64, Cread, Cread8, Cread16, Cread32, Cread64, Cursor, Cwrite
(D)
Data, Dcd, 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, Declare, Decrement, Define, Delayms, Delayus, DelayCs, Device, Dig, Dim, Djc, Djnc, Djnz, Djz, Dt, DTMfout, Dw, Dword, Double, dSin, dCos, dTan, dExp, dLog, dLog10, dAtan, dAtan2, dAsin, dAcos, dSqr, dAbs
(E)
Edata, Eeprom_Size, Else, ElseIf, End, EndAsm, EndIf, EndM, EndSelect, EndProc, 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, fAbs
(G)
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, Hig, HighLow_Tris_Reverse, Hpwm, Hrsin, Hrsin1, Hrsin2, Hrsin3, Hrsin4, Hrsout, Hrsout1, Hrsout2, Hrsout3, Hrsout4, Hserin, Hserin1, Hserin2, Hserin3, Hserin4, Hserout, Hserout1, Hserout2, Hserout3, Hserout4, Hserial_Baud, Hserial1_Baud, Hserial2_Baud, Hserial3_Baud, Hserial4_Baud, Hserial_Clear, Hserial1_Clear, Hserial2_Clear, Hserial3_Clear, Hserial4_Clear, Hserial_Parity, Hserial1_Parity, Hserial2_Parity Hserial3_Parity, Hserial4_Parity,
(I)
I2C_Bus_SCL, I2C_Slow_Bus, I2Cin, I2Cout, I2CWrite, I2CRead, ICD_Req, ICos, If, Ijc, Ijnc, Ijnz, Ijz, Inc, Include, Increment,Inkey, Input, Internal_Bus, Internal_Font, IntToFix16_8, IntToFix8_8, IrIn, IrIn_Pin, ISin, ISqr

(K)

Keyboard_CLK_Pin, Keyboard_DTA_Pin, Keyboard_IN, Keypad_Port

(L)

Label_Word, 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, Log, Log10, LookDown, LookDownL, LookUp, LookUpL, Low, Lread, Lread8, Lread16, Lread32, Lread64

(M)

Macro_Params, Max, Mid$, Min, Mouse_CLK_Pin, Mouse_Data_Pin, Mouse_In, Movlw, Mssp_Type

(N)

Ncd, Next, Nop, Num_Bit, Num_Byte, Num_Dword, Num_Float, Num_Word, Num_Wreg

(O)

Onboard_Adc, Onboard_USART, Onboard_Usb, Optimiser_Level, Oread, Org, Output, Owin, Owout, Owrite, OSC_PLLDIV

(P)

Pause, Pauseus, Pixel, Plot, 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, Proton24_Start_Address, PulsIn, PulseIn, Pulsin_Maximum, PulseOut, Push, Pwm, Ptr8, Ptr16, Ptr32, Ptr64, Proc

(R)

Random, RC5in, RC5in_Extended, RC5in_Pin, RCall, RCin, RcTime, Read, Rem, Remarks, Reminders, Rep, Repeat, Res, Retfie, Retlw, Return, Return_Type, Return_Var, Rev, Right$, Rol, Ror, Rsin, Rsin_Mode, Rsin_Pin, Rsin_Timeout, Rsout, Rsout_Baud, Rsout_Mode, Rsout_Pace, Rsout_Pin, Return_Bit, Return_Byte, Return_Dword, Return_Float, Return_Word, Return_Wreg

(S)

SCL_Pin, SDA_Pin, Seed, Select, Serial_Baud, Serial_Data, Serial_Parity, Serin, Serout, Servo, Set, SetBit, Shift_DelayUs, ShiftIn, Shin, Shout, Show_Expression_Parts, Show_System_Variables, Signed_Dword_Terms, Sin, SizeOf, Sleep, Slow_Bus, Small_Micro_Model, Snooze, SonyIn, SonyIn_Pin, Sound, Sound2, Sqr, Stack_Size, Step, Stop, Str, Str$, Str$, StrCmp, String, Strn, Swap, Symbol, Setup_PLL

(T)

Tan, Then, To, Toggle, ToLower, Toshiba_Command, Toshiba_UDG, ToUpper, Touch_Active, Touch_Read, Touch_HotSpot, Touch_HotSpotTable, Trim, TrimLeft, TrimRight

(U)

Udata, UnPlot, Until, Upper

(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, Write_OSCCON, Write OSCCONL, Write_OSCCONH

(X)

Xtal

_adc, _adcres, _code, _core, _defined, _device, _eeprom, _flash, _mssp, _ports, _ram, _type, _usart, _usb, _xtal

Notes.