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

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

PICmicro™ is a trade name of Microchip Technologies Inc.	www.microchip.com
Proton™ is a trade name of Crownhill Associates Ltd.	www.crownhill.co.uk
EPIC™ is a trade name of microEngineering Labs Inc.	www.microengineeringlabs.com
The Proton IDE was written by David Barker of Mecanique	www.mecanique.co.uk
Proteus VSM © Copyright Labcenter Electronics Ltd 2004	www.labcenter.co.uk

Web url’s correct at time of publication.

The Proton compiler and documentation was written by Les Johnson

All Manufacturer Trademarks Acknowledged.

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

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

Introduction

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

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

Contact Details

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

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

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

Telephone
(+44) 01353 749990

Fax
(+44) 01353 749991

Email
sales@crownhill.co.uk

Web Sites
http://www.crownhill.co.uk
http://www.picbasic.org

Table of Contents.

Proton IDE Overview

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

Code Explorer

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

Compiler Results

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

Programmer Integration

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

Integrated Bootloader

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

Real Time Simulation Support

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

Serial Communicator

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

Online Updating

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

Plugin Architecture

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

Supported Operating Systems

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

Minimum Hardware Requirements

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

Menu Bar

File Menu

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

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

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

- **Save As** - Displays a save as dialog, enabling you to name and save a document to disk.

- **Close** - Closes the currently active document.

- **Close All** - Closes all editor documents and then creates a new editor document.

- **Reopen** - Displays a list of Most Recently Used (MRU) documents.

- **Print Setup** - Displays a print setup dialog.

- **Print Preview** - Displays a print preview window.

- **Print** - Prints the currently active editor page.

- **Exit** - Enables you to exit the Proton IDE.

Edit Menu

- **Undo** - Cancels any changes made to the currently active document page.

- **Redo** - Reverse an undo command.

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

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

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

- **Delete** - Deletes any selected text. This option is disabled if no text has been selected.

- **Select All** - Selects the entire text in the active document page.

- **Change Case** - Allows you to change the case of a selected block of text.

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

View Menu

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

Help Menu

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

Main Toolbar

 **New**

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

 **Open**

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

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

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

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

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

 **Undo**
Cancels any changes made to the currently active document page.

 **Redo**
Reverse an undo command.

 **Print**
Prints the currently active editor page.

Edit Toolbar

 **Find**
Displays a find dialog.

 **Find and Replace**
Displays a find and replace dialog.

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

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

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

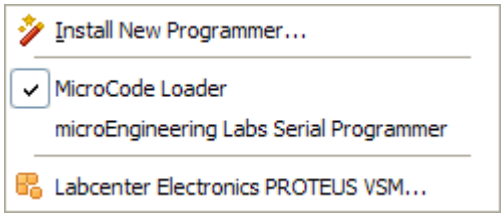
Block Uncomment
Removes the comment character from each line of a selected block of text. If multiple lines are not selected, a single comment is removed from the start of the line containing the cursor.

Compile and Program Toolbar

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

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

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



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

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

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

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

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

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

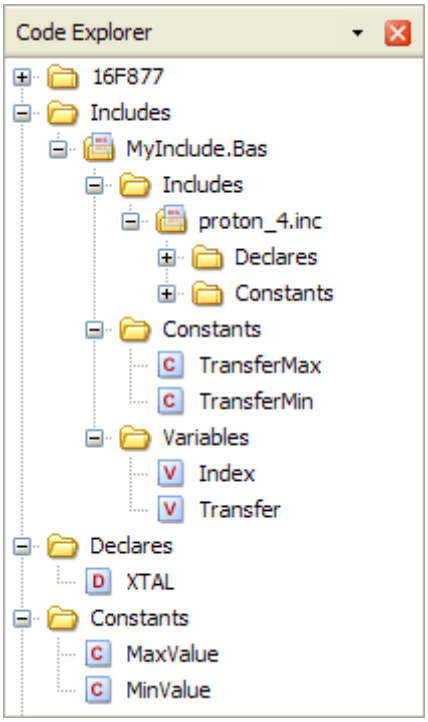
Device Node
The device node is the first node in the explorer tree. It displays your currently selected processor type. For example, if you program has the declaration: -

```
Device = 16F877
```

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

Include File Node

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

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

History back button
History forward button

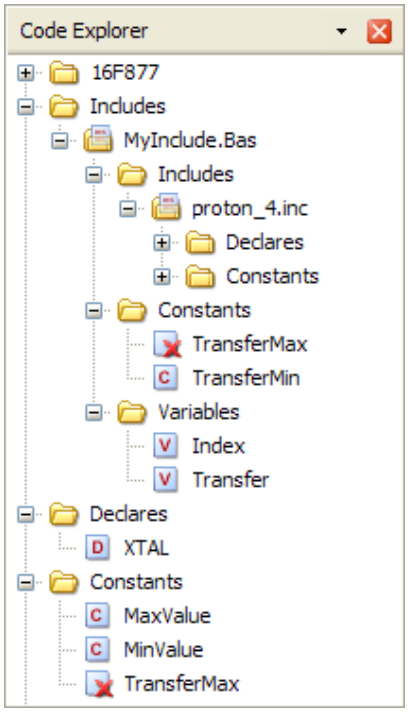
Additional Nodes
Declares, constants, variables, alias and modifiers, labels, macros and data label explorer nodes work in much the same way. Clicking on any of these nodes will take you to its declaration. If you want to find the next occurrence of a declaration, you should enable automatically select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

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

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

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

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



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

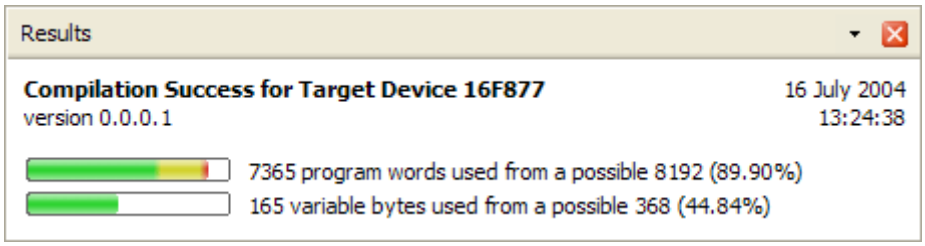
Some features of the compiler of not available for some MCU types. For example, you cannot have a string declaration when using a 14 core part (for example, the 16F877). If you try to do this, the explorer node icon changes from its default state and displays a . You will also see this icon displayed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

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

Results View

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

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



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

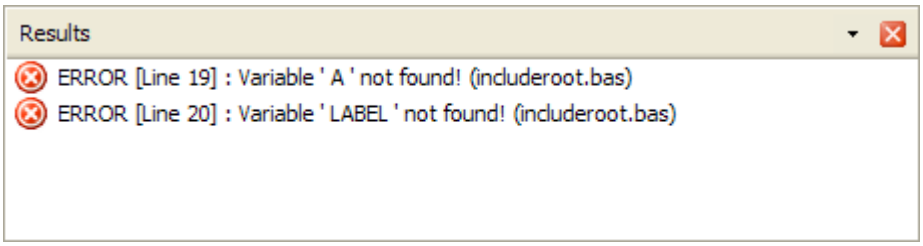
Version Numbers
The version number is automatically incremented after a successful build. Version numbers are displayed as major, minor, release and build. Each number will rollover if it reaches 256. For example, if your version number is 1.0.0.255 and you compile again, the number displayed will be 1.0.1.0. You might want to start 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 messages during compilation. Because you should not normally ignore warning messages, the IDE will always display the error view, rather than the success view, if warnings have been generated.

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

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



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

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

Editor Options

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

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

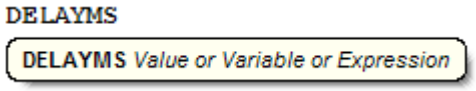
Show Right Gutter
Displays a line to the right of the main editor. You can also set the distance from the left margin (in characters). This feature can be useful for aligning your program comments.

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

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

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

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



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

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

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

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

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

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

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

Dim MyIndex **as Byte**

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

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

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

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

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

Highlighter Options

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

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

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

Reserved Word Formatting

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

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

Uppercase - the IDE will display the keyword in uppercase.

Lowercase - the IDE will display the keyword in lowercase.

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

Please note that the actual keyword text is not physically changed, it just changes the way it is displayed in the editor window. For example, if you save your document and load it into 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).

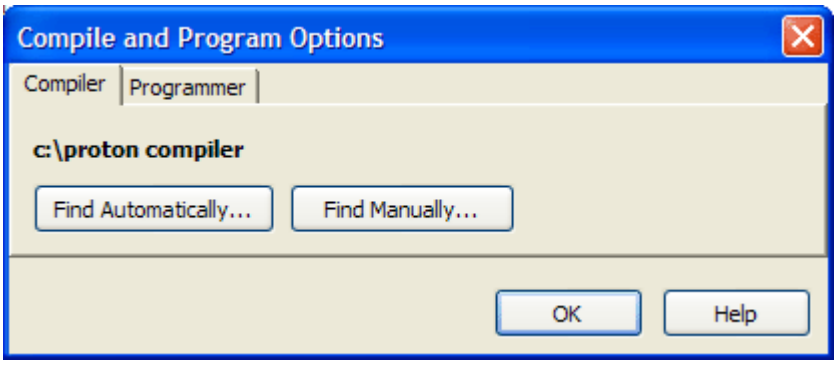
Header options allows you to change the author and copyright name that is placed in a header when a new document is created. For example: -

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

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

Compile and Program Options

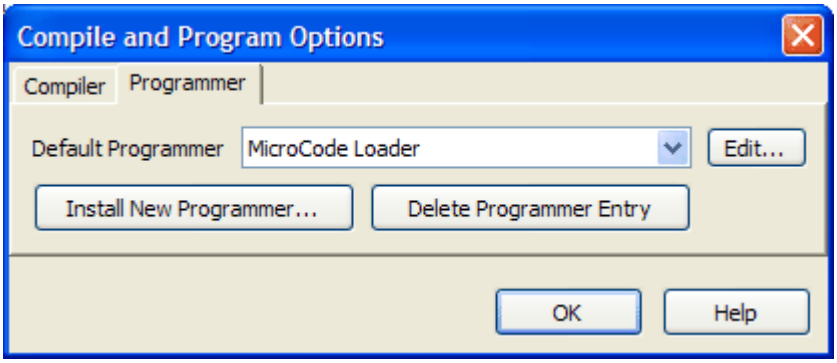
Compiler Tab



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

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

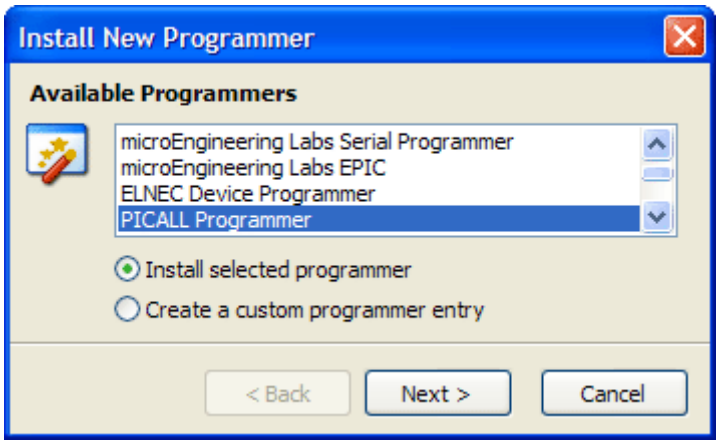
Programmer Tab



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

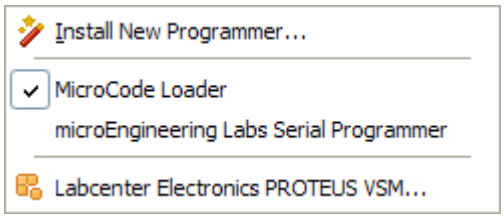
Installing a Programmer

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



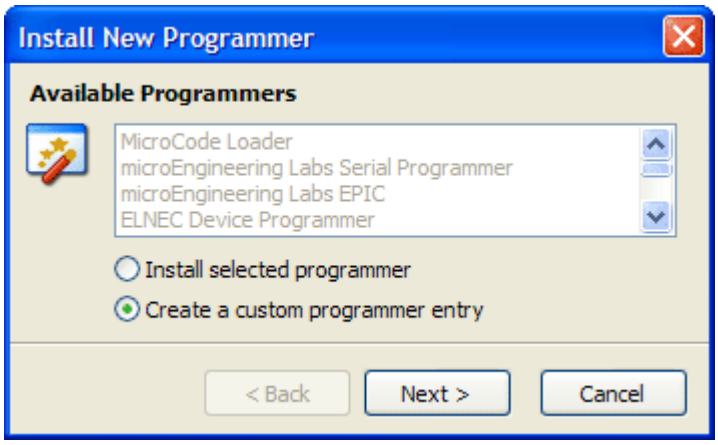
Select the programmer you want Proton IDE to use, then choose the Next button. Proton IDE will now search your computer until it locates the required executable. If your programmer is not in the list, you will need to create a custom programmer entry.
Your programmer is now ready for use. When you press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



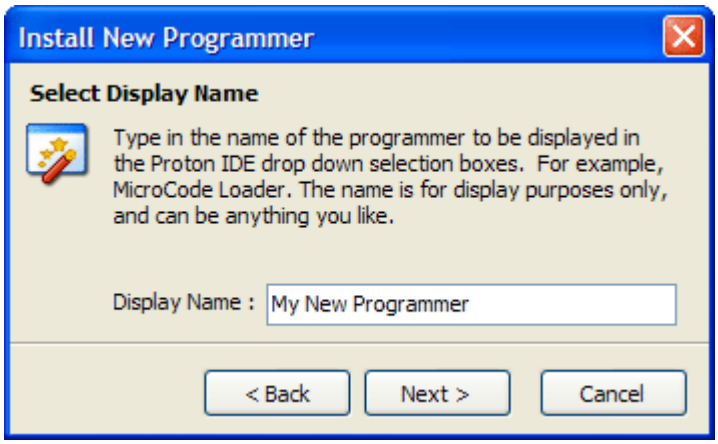
Creating a custom Programmer Entry

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



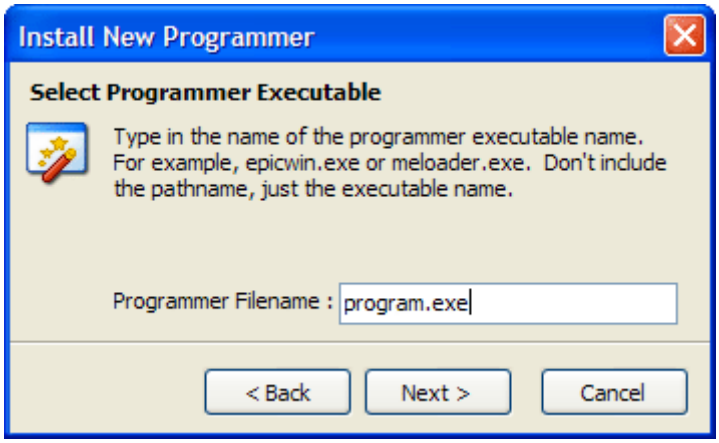
Select Display Name

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



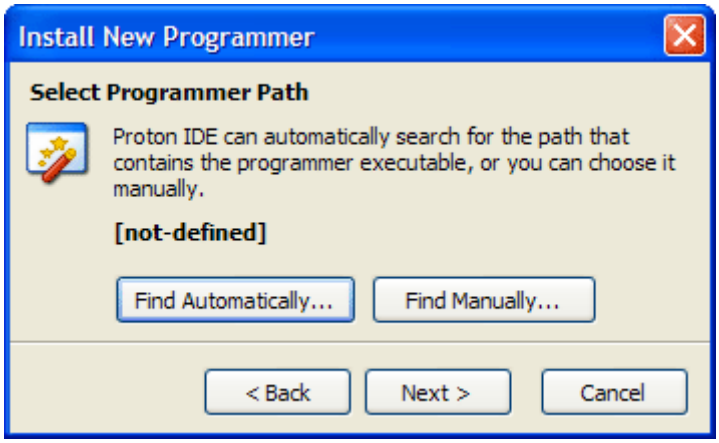
Select Programmer Executable

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



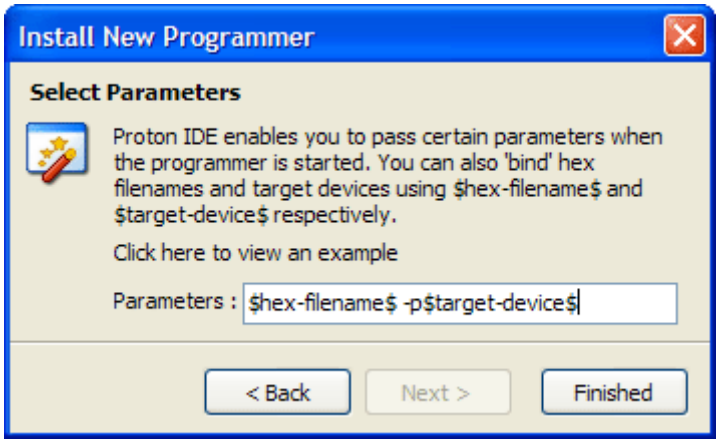
Select Programmer Path

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



Select Parameters

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



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

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

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

```
-pPIC16F628 blink.hex
```

Parameter Summary

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

Microcode Loader

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

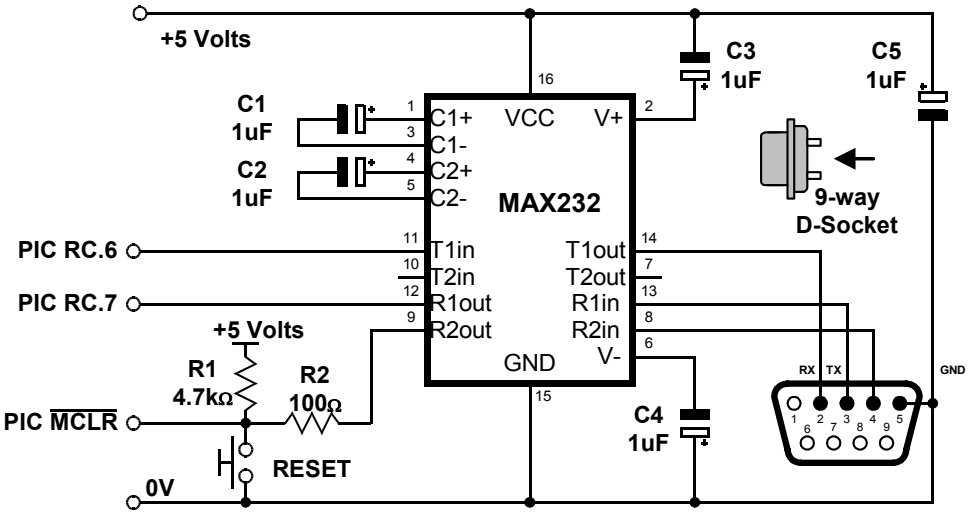
When power is first applied to the microcontroller (or it is reset), the bootloader first checks to see if the MicroCode Loader application has something for it to do (for example, program your code into the target device). If it does, the bootloader gives control to MicroCode Loader until it is told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

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

Hardware Requirements

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

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



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

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

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

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

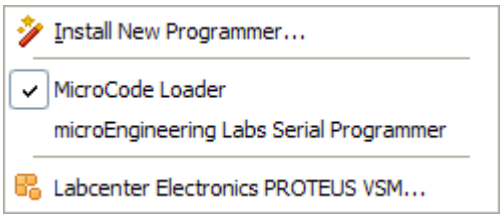
Programming the Loader Firmware

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

Configuring the Loader

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

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



Using MicroCode Loader

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

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

Loader Options

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

Program Code

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

Program Data

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

Verify Code When Programming

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

Verify Data When Programming

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

Verify Code

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

Verify Data

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

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

Run User Code After Programming
Exit the bootloader process immediately after programming and then start running the target user code. The default is On.

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

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

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

Loader Main Toolbar

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

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

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

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

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

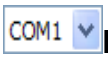
 **Verify**
The verify button will compare the currently loaded *.hex file code and eeprom data with the code and eeprom data located on the target microcontroller. When verifying the target device, both user code and eeprom data are verified by default. However, you may want to just verify code or eeprom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

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

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

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

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



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

IDE Plugins

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

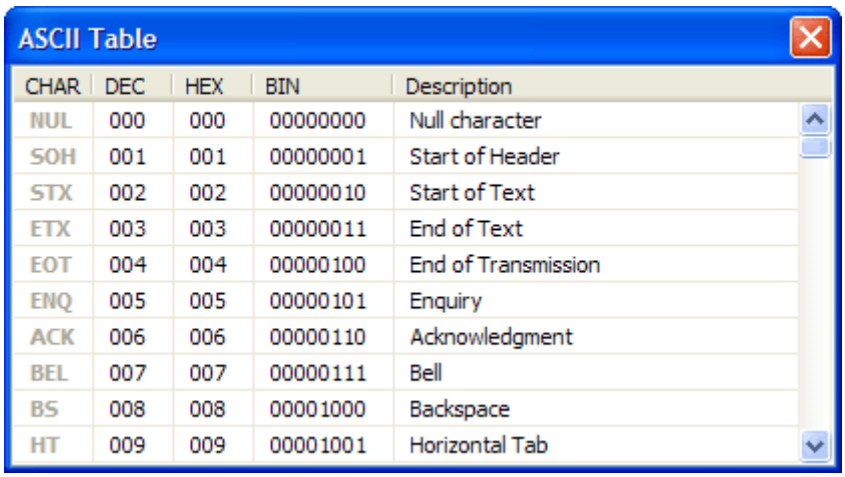
ASCII Table
Assembler
Hex View
Serial Communicator
Labcenter Electronics Proteus VSM

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

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

ASCII Table

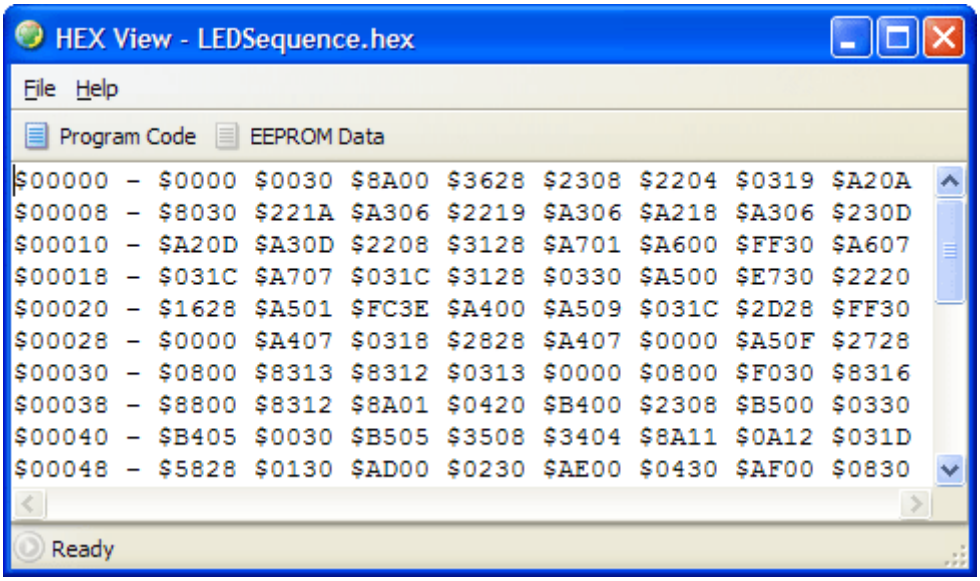
The American Standard Code for Information Interchange (ASCII) is a set of numerical codes, with each code representing a single character, for example, 'a' or '$'.



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.

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

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

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

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

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

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

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

Serial Communicator

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

Menu options

File Menu

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

Edit Menu

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

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

View Menu

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

Help Menu

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

Serial Communicator Main Toolbar

 **Clear**
Clears the contents of either the transmit or receive window.

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

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

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

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

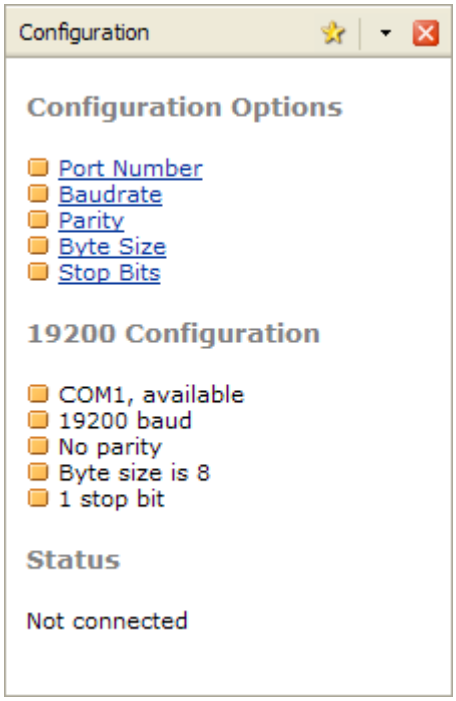
 **Paste**
Paste the contents of the clipboard into either the transmit or receive window. This option is disabled if the clipboard does not contain any suitable text.

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

 **Disconnect**
Disconnect the Serial Communicator from a serial port.

Configuration

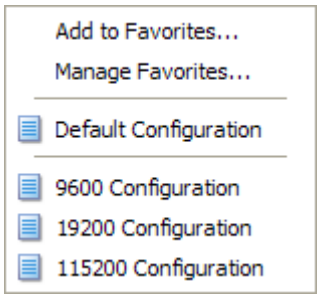
The configuration window is used to select the COM port you want to connect to and also set the correct communications protocols.



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

Favourites

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



Add to Favourites

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

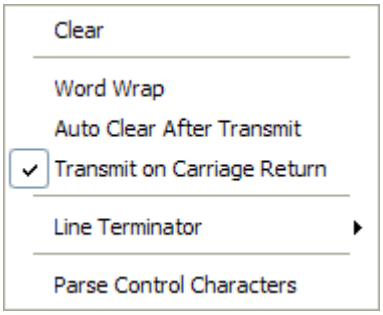
Manage Favourites

Select this option to remove a previously saved configuration favourite.

Notes

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

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



Clear
Clear the contents of the transmit window.

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

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

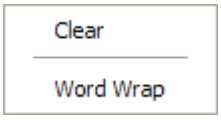
Transmit on Carriage Return
This option will automatically transmit data when the carriage return key is pressed. If this option is disabled, you will need to manually press the send button or press F4 to transmit.

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

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

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

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



Clear
Clear the contents of the receive window.

Word Wrap
When enabled, incoming data is automatically word wrapped.

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

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

The Proton Plus Development Suite comes shipped with a free demonstration version of the Proteus simulation environment and also a number of pre-configured Virtual Hardware Boards (VHB). Unlike the professional version of Proteus, you are unable to make any changes to the pre-configured boards or create your own boards.
If you already have a full version of Proteus VSM installed on your system (6.5.0.5 or higher), then this is the version that will be used by the IDE. If you don't have the full version, the IDE will default to using the demonstration installation.

System Requirements
Windows XP or Vista
512MB RAM (1 GB or higher recommended)
500 MHz Processor

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

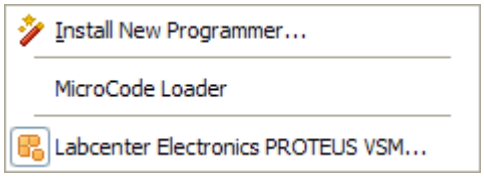
ISIS Simulator Quick Start Guide
This brief tutorial aims to outline the steps you need to take in order to use Labcenter Electronics Proteus Virtual System Modelling (VSM) with the Proton IDE. The first thing you need to do is load or create a program to simulate. In this worked example, we will keep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

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

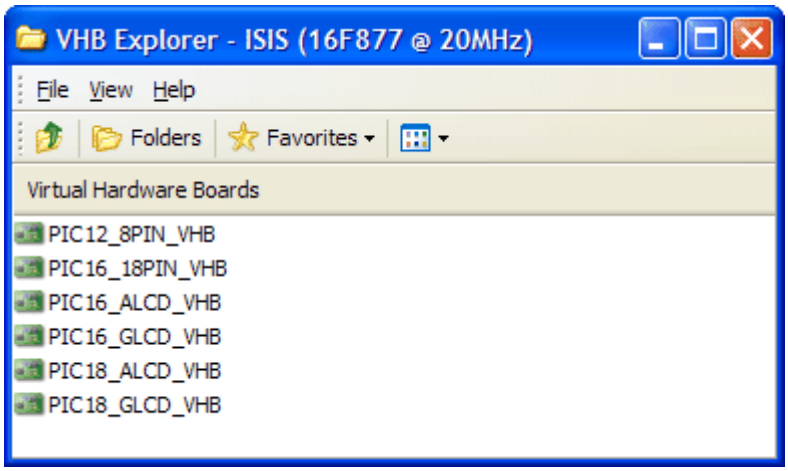


```
Delayms 500
Low LED
Delayms 500
Goto MainProgram
```

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



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



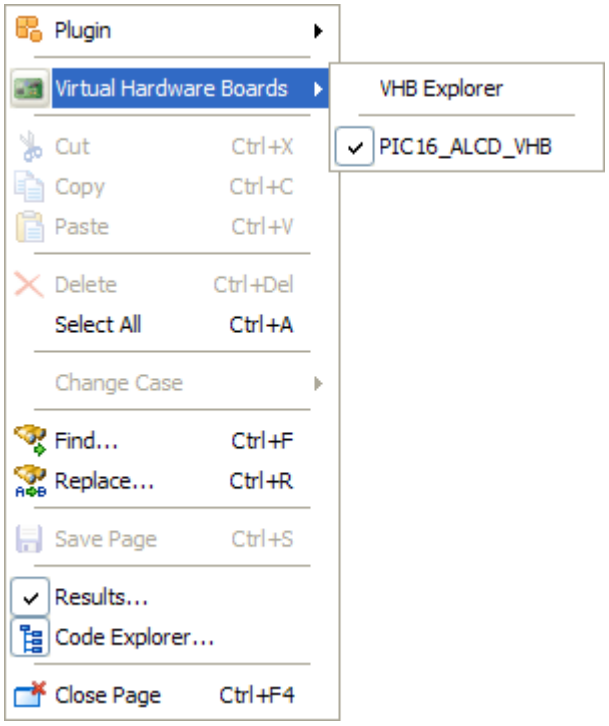
VHB Explorer is the IDE plugin that co-ordinates activity between the IDE and the simulator. Its primary purpose is to bind a Virtual Hardware Board to your program. In this example, the program has been built for the 16F877 MCU which flashes an LED connected to PortD.0. To run the simulation for this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

Additional Integration Tips

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

Virtual Hardware Boards Favourites

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



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

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

Compiler Overview.

PICmicro Devices

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

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

Limited 12-bit Device Compatibility.

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

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

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

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

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

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

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

Print, Rsout, Busin, Busout

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

Programming Considerations for 12-bit core Devices.

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

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

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

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

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

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

Device Specific issues

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

```
CMCON = 7
```

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

```
Declare All_Digital = True
```

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

Alternatively, you can manipulate the hardware registers directly: -

```
ADCON1 = 7
```

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

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

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

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

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

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

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

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

Some devices have internal pull-up resistors on PortB, or GPIO. These may be enabled or disabled by issuing the **PortB_Pullups** command: -

```
  Declare PortB_Pullups = On     ' Enable PortB pull-up resistors
or
  Declare PortB_Pullups = Off    ' Disable PortB pull-up resistors
```

Identifiers

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

Line Labels

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


Variables

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

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

Dim *Label* **as** *Size*

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

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

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

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

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

Intuitive Variable Handling.

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

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

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

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

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

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

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

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

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

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

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

- **Bit** type variables may hold a 0 or a 1. These are created 8 at a time, therefore declaring a single **Bit** type variable in a program will not save RAM space, but it will save code space, as **Bit** type variables produce the most efficient use of code for comparisons etc.

- **Byte** type variables may hold an unsigned value from 0 to 255, and are the usual work horses of most programs. Code produced for **Byte** sized variables is very low compared to signed or unsigned **Word**, **DWord** or **Float** types, and should be chosen if the program requires faster, or more efficient operation.

- **SByte** type variables may hold a 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 varsatile of the variable family types. However, more so than **Dword** types, this comes at a price as floating point expressions and comparisons will use more code space within the PICmicro™. Use this type of variable sparingly, and only when strictly necessary. Smaller floating point values usually offer more accuracy.

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

Floating Point Math

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

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

```
Dim Flt as Float
```

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

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

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

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

Floating Point Format

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

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

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

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

IEEE-754 to Microchip

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

Microchip to IEEE-754

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


Variables Used by the Floating Point Libraries.

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

Floating Point Example Programs.

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

  Cls
  Flt = FlNum * 10
  Print Dec Flt
  Stop

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

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

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

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

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


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

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

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

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

Code space requirements.

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

Floating Point To Integer Rounding

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

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

The variable DwordVar will hold the value of 3.

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

Floating Point Exception Flags

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

```
Dim _FP_FLAGS as Byte System
```

The exceptions are:

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

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

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

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

More Accurate Display or Conversion of Floating Point values.

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

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

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

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

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

Aliases

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Places Var1 first, then Var2: -

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

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

Finer points for variable handling.

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

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

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

```
Dim Wrd as Word
```

Will produce the assembler code: -

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

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

```
WrdH = 1
```

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

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

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

```
Dim Dwd as Dword
```

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

```
Dwd.Byte3 = 1
```

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

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

```
Dim sDwd as SDword  ' Declare a 32-bit signed variable

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


Constants

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

Dim *Label* **as** *Constant expression*

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

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

Symbols

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

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

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

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

Floating point constant can also be created using expressions.

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

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

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

Symbol is also useful for aliasing Ports and Registers: -

```
Symbol LED = PORTA.1       ' LED now references bit-1 of PortA
Symbol T0IF = INTCON.2     ' T0IF now references bit-2 of INTCON register
```

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

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


Numeric Representations

The compiler recognises several different numeric representations: -

Binary is prefixed by %. i.e. %0101
Hexadecimal is prefixed by $. i.e. $0A
Character byte is surrounded by quotes. i.e. "a" represents a value of 97
Decimal values need no prefix.
Floating point is created by using a decimal point. i.e. 3.14

Quoted String of Characters

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

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

\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"
```

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

Null Terminated

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

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

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

Ports and other Registers

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

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

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


The compiler can also combine16-bit registers such as TMR1 into a **Word** type variable. Which makes loading and reading these registers simple: -

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

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

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

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

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

General Format

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

Multiple instructions and labels can be combined on the same line by separating them with co-lons '**:**'.

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

Multiple-line version: -

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

Single-line version: -

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


A Typical basic Program Layout

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

The recommended layout for a program is shown below.

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

   GoTo Main                          ' Jump over the subroutines (if any)

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

For example:

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

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

Line Continuation Character '_'

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

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

or

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


Creating and using Arrays

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

An array is defined using the following syntax: -

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

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

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

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

PICmicro™ Memory Map Complexities.

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

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

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

or

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

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

Ignoring this warning will spell certain failure of your program.

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

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

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

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

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

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

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

18F and enhanced 14-bit core device simplicity.

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

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

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

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

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

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

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

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

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

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

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


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

Using Arrays in Expressions.

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

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

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

(10 * 20) / 20

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

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

Arrays as Strings

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Creating and using Strings

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

The syntax to create a string is : -

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

String Name can be any valid variable name. See **Dim** .
String Length can be any value up to 255, allowing up to 255 characters to be stored.

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

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

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

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

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

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

```
Print DestString
```

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

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

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

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

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

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

Cstr Example

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

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

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

More efficient Example of above code

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

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

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

Estr Example

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

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

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


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

Str$ Example

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

  Device = 18F4520                  ' A suitable device for Strings
  Dim DestString as String * 30     ' Create a String capable of holding 30
characters
  Dim SourceString as String * 20   ' Create another String
  Dim Wrd1 as Word                  ' Create a Word variable

  Wrd1 = 1234                       ' Load the Word variable with a value
  SourceString = "Value = "         ' Load the string with characters
  DestString = SourceString + Str$(Dec Wrd1)  ' Concatenate the string
  Print DestString                ' Display the result which is "Value = 1234"
  Stop
```

Left$ Example

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

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

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

Right$ Example

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

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

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

Mid$ Example

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

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

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


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

ToUpper Example

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

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

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

ToLower Example

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

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

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

Loading a String Indirectly

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

Example

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

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

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


Slicing a String.

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

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

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

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

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

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

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

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

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

The example below shows a more practical String slicing demonstration.

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

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


Notes

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

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

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

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

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

Creating and using Code Memory Strings

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

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

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

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

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

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

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

```
  Print Cstr String1
```

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

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

First the standard way of displaying text: -

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

Now using the **Cstr** modifier: -

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

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


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

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

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

Example 1

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

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

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

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

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

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

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

  Cls                   ' Clear the LCD

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


Creating and using Eeprom Memory Strings with Edata

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

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

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

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

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

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

```
Print Estr String1
```

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

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

First the standard way of displaying text: -

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

Now using the **Estr** modifier: -

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

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

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

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

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

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

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

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

Notes

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

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

String Comparisons

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

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

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

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

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

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

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

Example 1

```
' Simple string variable comparison

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

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

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

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

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

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

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

Example 2

```
' String variable to Quoted character string comparison

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

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

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

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

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

Example 3

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

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

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

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


Example 4

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

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

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

CodeString:
  Cdata "EGGS", 0
```

Example 5

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

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

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

Relational Operators

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

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

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

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

Boolean Logic Operators

The **If-Then-Else-Endlf**, **While-Wend**, and **Repeat-Until** conditions now support the logical operators **not**, **and**, **or**, and **xor**. The **not** operator inverts the outcome of a condition, changing false to true, and true to false. The following two **If-Then** conditions are equivalent: -

```
  If Var1 <> 100 Then NotEqual     ' Goto notEqual if Var1 is not 100.
  If not Var1 = 100 Then NotEqual  ' Goto notEqual if Var1 is not 100.
```

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

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

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

Parenthesis (or rather the lack of it!).

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

```
  If (Var1 + 3) = 10 Then do something.    Is allowed.
```

but: -

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

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

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

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

Then operand always required.

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

