PureBasic Reference Manual
6.21

http://www.purebasic.com/

June 11, 2025

Contents

Part I

General

Chapter 1

Introduction

PureBasic is an "high-level" programming language based on established "BASIC" rules. It is mostly compatible with any other "BASIC" compiler. Learning PureBasic is very easy! PureBasic has been created for beginners and experts alike. Compilation time is extremely fast. This software has been developed for the Windows operating system. We have put a lot of effort into its realization to produce a fast, reliable and system-friendly language.
The syntax is easy and the possibilities are huge with the "advanced" functions that have been added to this language like pointers, structures, procedures, dynamic lists, maps, interfaces, modular programmation (modules), an inline assembler and much more. For the experienced coder, there are no problems gaining access to any of the legal OS structures or API objects.
PureBasic is a portable programming language which currently works on Linux, Mac OS X and Windows computer systems and machines running Raspberry PI OS. This means that the same code can be compiled natively for the OS and use the full power of each. There are no bottlenecks like a virtual machine or a code translator, the generated code produces an optimized executable.

The main features of PureBasic

- Intel/AMD and others' x86 and x64, M1, M2, M3, arm32 and arm64 support
- 86 native libraries
- Hundreds of functions
- Built-in arrays, dynamic lists, complex structures, maps, pointers and variable definitions
- Supported types: Byte (8-bit), Word (16-bit), Long (32-bit), Quad (64-bit), Float (32-bit), Double (64-bit) and Characters
- User defined types (structures)
- Built-in string types (characters), including ascii and unicode
- Powerful macro support
- Constants, binary and hexadecimal numbers supported
- Expression reducer by grouping constants and numeric numbers together
- Standard arithmetic support in respect of sign priority and parenthesis: +, -, /, *, and, or, «, »
- Extremely fast compilation
- Procedure support for structured programming with local and global variables
- All Standard BASIC keywords: If-Else-EndIf, Repeat-Until, etc
- Specialized libraries to manipulate BMP pictures, windows, gadgets, DirectX, etc
- Specialized libraries are very optimized for maximum speed and compactness
- The Win32 API is fully supported as if they were BASIC keywords
- Inline Assembler
- Precompiled structures with constants files for extra-fast compilation
- Integrated debugger to follow the execution of a program and correct errors more easily
- Configurable CLI compiler
- Dedicated editor with automatically highlighted syntax
- Very high productivity, comprehensive keywords, online help

- System friendly, easy to install and easy to use
- IDE and help available in english, french and german
- SDK Visual C
- Compilation of windowed program, console and DLL
- Creation of integrated wysiwyg windows
- Creation of windows with automatic reorganization of gadgets (layout) with the 'Dialog' library
- 33 native gadgets
- 1 gadget specialized in OpenGL
- 1 scintilla gadget
- 1 webview gadget to easily create HTML/CSS based user interface
- Process, thread, mutex, semaphore
- Drag'n drop
- DPI for MS Windows and OSX
- QT and GTK3 for Linux
- DirectX and OpenGL systems
- Import of static (lib) or dynamic libraries (dll, so, etc.)
- Drawing with antialiasing with the 'Vector' library
- 3D functions with the 3D engine OGRE
- Pseudotypes: p-ascii, p-utf8, p-bstr, p-unicode, p-variant
- Array, linked-list (list), maps, Database (MySQL, SQLite, ODBC, PostgreSQL, Maria)
- json, xml
- Regular expression
- http, ftp, mail, application server / client, CGI and FastCGI
- Cypher
- OnError
- Compressor/decompressor: BriefLZ, JCALG1, LZMA, Tar, Zip
- ini file
- Printer, serial port
- Runtime

Here is the exhaustive list of all the libraries that PureBasic offers:

```
  Windows & system :
    arrays
    cgi
    cipher
    clipboard
    console
    database
    date
    debugger
    desktop
    dialog
    dragdrop
    file
    filesystem
    font
    ftp
    gadget
    help
    http
    json
    library
    lists
    mail
    maps
    math
    memory
    menu
    network
    onerror
```

```
  packer
  preference
  printer
  process
  regularexpression
  requester
  runtimes
  scintilla
  serialport
  sort
  statusbar
  string
  system
  systray
  thread
  toolbar
  webview
  window
  xml

2D & Media :
  2ddrawing
  audiocd
  font
  image
  imageplugin
  joystick
  keyboard
  mouse
  movie
  music
  screen
  sound
  soundplugin
  sprite
  vectordrawing

3D Engine :
  billboard
  camera
  engine3d
  entity
  entityanimation
  gadget3d
  joint
  light
  material
  mesh
  node
  nodeanimation
  particle
  skeleton
  sound3d
  specialeffect
  spline
  staticgeometry
  terrain
  text3d
  texture
```

```
vehicle
vertexanimation
window3d
```

Chapter 2

Terms And Conditions

This program is provided "**AS IS**". Fantaisie Software are NOT responsible for any damage (or damages) attributed to PureBasic. You are warned that you use PureBasic at your own risk. No warranties are implied or given by Fantaisie Software or any representative.
The demo version of this program may be freely distributed provided all contents, of the original archive, remain intact. You may not modify, or change, the contents of the original archive without express written consent from Fantaisie Software.
PureBasic has an user-based license. This means you can install it on every computer you need but you can't share it between two or more people.
All components, libraries, and binaries are copyrighted by Fantaisie Software. The PureBasic license explicitly forbids the creation of DLLs whose primary function is to serve as a 'wrapper' for PureBasic functions.
Fantaisie Software reserves all rights to this program and all original archives and contents.
PureBasic uses the Scintilla editing component and its related license can be consulted here .
PureBasic uses GCC compiler and its related license can be consulted here and here
The PureBasic uses FASM flat assembler and its related license can be consulted here
The PureBasic uses PellesC toolchain and its related license can be consulted here
PureBasic built-in libraries uses opensources components, the licenses can be consulted here and here

Chapter 3

System requirements

PureBasic will run on Windows Vista, Windows 7, Windows 8, Windows 10 and Windows 11 (in both 32-bit and 64-bit edition), Linux (kernel 2.2 or above), MacOS X (10.15 or above) and Raspberry PI OS (bookworm/debian 12 or above).
If there are any problems, please contact us: support@purebasic.com

Windows

For many graphic functions like 3D or Sprites there is needed the DirectX9.0c version. Windows Vista and later don't have installed this from start, so it need to be installed manually.
You can download and install it via the "DirectX End-User Runtime Web Installer" here: http://www.microsoft.com/en-us/download/details.aspx?displaylang=en&id=35.

Linux

Required packages to use PureBasic:

- sdl 1.2 devel (for 'Joystick' and 'AudioCD' libraries)
- gtk 3 (For GUI programs)
- libstdc++ devel
- gcc correctly installed
- iodbc and iodbc-devel to be able to use the Database commands (see www.iodbc.org)
- libwebkit should be installed to have the WebGadget() working.
- libvlc-devel for the Movie commands Check the INSTALL and README file for more information.

MacOS X

You need to install the Apple developer tools to be able to use PureBasic. The tools can be found either on one of the install CDs or on the Apple web site: http://developer.apple.com/
Be aware to use the latest version of the tools (e.g. XCode), which is needed for your OS version! The commandline tools needs to be installed from XCode once installed.

Chapter 4

Installation

To install PureBasic, just click on the install wizard, follow the steps, and then click on the PureBasic icon (found on the desktop or in the start-menu) to launch PureBasic.
To use the command line compiler, open a standard command line window (CMD) and look in the "Compilers\" subdirectory for PBCompiler.exe. It's a good idea to consider adding the "PureBasic\Compilers\" directory to the PATH environment variable to make the compiler accessible from any directory.
Important note: to avoid conflicts with existing PureBasic installations (maybe even with user-libraries), please install a new PureBasic version always in its own new folder. See also the chapter Using several PureBasic versions .

Chapter 5

Order

PureBasic is a low-cost programming language. In buying PureBasic you will ensure that development will go further and faster. For personal use (i.e.: not used by a commercial organization) the updates are unlimited, unlike most other software out there. This means when you buy PureBasic you will get all future updates for free, on the web site. Better still, you get the four versions of PureBasic (MacOSX, Linux, Raspberry Pi, and Windows) for the same price! For ease of ordering, you can safely use our secure online method. Thanks a lot for your support!

The demo-version of PureBasic is limited as shown below:

- No DLLs can be created
- you can't use the whole external Win32 API support
- no development kit for external libraries
- maximum number of source lines: 800

Full version of PureBasic:

Price for full version: 79 Euros
There are special prices for company licences (499 Euros) and educational licenses (199 Euros for a school class).

Online Ordering

Payment online (www.purebasic.com) is available and very secure. (Note: Such orders are handled directly by Fred.)

If you live in Germany or Europe and prefer paying to bank account you can also send your registration to the German team member. In this case please send your order to following address:

```
André Beer
Siedlung 6
09548 Deutschneudorf
Germany
e-mail: andre@purebasic.com

Bank Account:
```

```
 Deutsche Kreditbank AG
 Account 15920010 - Bank code 12030000
 (For transactions from EU countries: IBAN: DE03120300000015920010 -
BIC/Swift-Code: BYLADEM1001)

 Paypal:
 andrebeer@gmx.de
 (This address can be used for Paypal transaction, if you want
personal contact to or an invoice from André.)
```

Delivering of the full version

The full version will be provided via your personal download account, which you will get on www.purebasic.com after successful registration. If you order from André, just write an e-mail with your full address or use this registration form and print it or send it via e-mail.

Chapter 6

Contact

Please send bug reports, suggestions, improvements, examples of source coding, or if you just want to talk to us, to any of the following addresses:

Frédéric 'AlphaSND' Laboureur

Fred 'AlphaSND' is the founder of Fantaisie Software and the main coder for PureBasic. All suggestions, bug reports, etc. should be sent to him at either address shown below:

s-mail :

Frédéric Laboureur
10, rue de Lausanne
67640 Fegersheim
France

e-mail : fred@purebasic.com

André Beer

André is responsible for the complete German translation of the PureBasic manual and website. PureBasic can be ordered in Germany also directly at him.
Just write an email with your full address (for the registration) to him. If needed you can also get an invoice from him. For more details just take a look here.
e-mail : andre@purebasic.com

Chapter 7

Acknowledgements

We would like to thank the many people who have helped in this ambitious project. It would not have been possible without them !

- All the registered users: To support this software... Many Thanks !

Coders

- **Timo 'Fr34k' Harter**: For the IDE, Debugger, many commands and the great ideas. PureBasic wouldn't be the same without him !
- **Benny 'Berikco' Sels**: To provide an amazing first class Visual Designer for PureBasic. That's great to have such kind of help !
- **Danilo Krahn**: To have done an huge work on the editor and on the core command set, without forget the tons of nice suggestions about code optimization and size reduction... Thanks a lot.
- **Sebastion 'PureFan' Lackner**: For the very nice OnError library, which adds professional error handling for final executables and several other commands !
- **Siegfried Rings**: Also for the very nice OnError library, as its a two men work :)
- **Marc Vitry**: To have kindly shared its SerialPort library sources which helped a lot for the native PureBasic library.
- **Stefan Moebius**: To have sent its DX9 subsystem sources, which have been the basis of the current DX9 subsystem for PureBasic.
- **Comtois**, "G-Rom" and "T-Myke": for the big help improving the 3D side of PureBasic ! That matters !
- **Gaetan Dupont-Panon**: For the wonderful new visual designer, which really rocks on Windows, Linux and OS X !

Miscellaneous

- **Roger Beausoleil**: The first to believe in this project, and his invaluable help with the main design and layout of PureBasic.
- **Andre Beer**: To spend time for improving the guides (including beginners guide) and do the complete translation into German. Big thanks!
- **Francis G. Loch**: To have corrected all the mistakes in the English guides ! Thanks again.
- **Steffen Haeuser**: For giving his valuable time, and assistance. For his invaluable explanations regarding aspects of PPC coding, and his very useful tips.
- **Martin Konrad**: A fantastic bug reporter for the Amiga version. To have done the forthcoming Visual Environment for PureBasic and some nice games.

- **James L. Boyd**: To have found tons of bugs in the x86 version and gave us tons of work :). Keep up the bug hunt !
- **Les**: For editing the English Fantaisie Software Web site & and this guide. This looks much better!
- **NAsm team**: To have done an incredible x86 assembler used to develop the PureBasic libraries
- **Tomasz Grysztar**: For FAsm, an excellent optimized x86 assembler currently used by PureBasic. http://flatassembler.net/
- **Pelle Orinius**: For the very good linker and resource compiler (from PellesC) currently used by PureBasic, and the invaluable and instant help when integrating the new linker in the PureBasic package. Thank you !
- **Jacob Navia**: For the linker (from the Lcc32 package) which was used in previous PureBasic versions
- **Thomas Richter**: For creating "PoolMem." A patch that decreases the compilation time by a factor of 2-3! Many thanks for allowing it to be added to the main archive.
- **Frank Wille**: For allowing the use of your excellent assemblers: "pasm," and "PhxAss". For all your tips, and invaluable assistance, in their usage. To help us a lot about debugging PowerPC executables.
- **Rings, Paul, Franco and MrVain/Secretly**: For extensive bug reports and help us to have a compiler as reliable and bug free as possible.
- **David 'Tinman' McMinn**: To enhance greatly the help and give the opportunity to the world to really discover PureBasic !
- **David 'spikey' Roberts**: To help creating the PureBasic beginners guide. Thanks a lot!
- **Leigh**: To have created and hosted the first PureBasic forums during 2 years. It has been a great amount of work, especially when the community started to grow. Thanks again !
- **Jean R. VIALE**: To have greatly improved the french documentation, and still doing it !

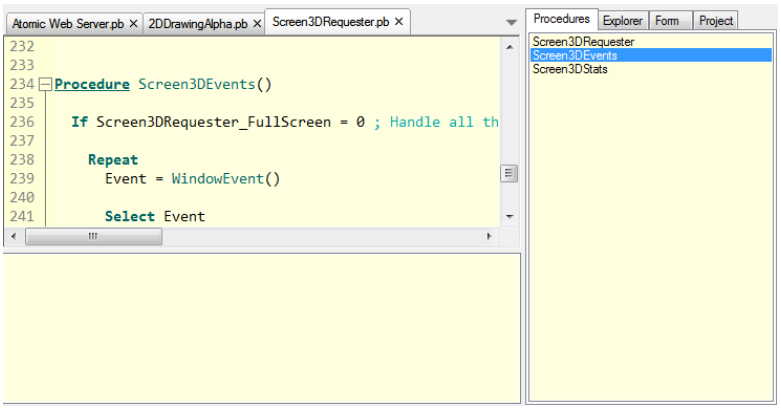
Part II

The PureBasic Editor

Chapter 8

Getting Started

The PureBasic IDE allows you to create and edit your PureBasic source codes, as well as run them, debug them and create the final executable. It has both an interface to the PureBasic Compiler , as well to the PureBasic Debugger .
The IDE main window contains of 3 major parts:



The code editing area (below the toolbar)
Here all the source codes are displayed. You can switch between them with the tabs located right above it.
The tools panel (on the right side by default)
Here you have several tools to make coding easier and increase productivity. The tools displayed here can be configured, and it can even be completely removed. See Customizing the IDE for more information.
The error log (located below the editing area)
In this area, the compiler errors and debugger messages are logged. It can be hidden/shown for each source code separately.

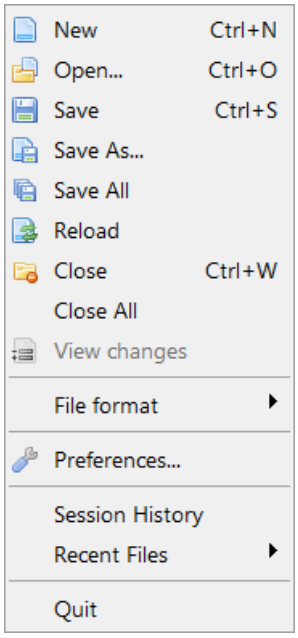
Other then that, there is the main menu and the toolbar. The toolbar simply provides shortcuts to menu features. It can be fully customized. To find out what each button does, move your mouse over it and wait until a small tool-tip appears. It shows the corresponding menu command. The menu commands are explained in the other sections.

Chapter 9

Working with source files

The file menu allows you to do basic file operations like opening and saving source codes.
You can edit multiple source code files at the same time. You can switch between them using the panel located under the Toolbar. Also the shortcut keys Ctrl+Tab and Ctrl+Shift+Tab can be used to jump to the next or previous open source file, respectively.
The IDE allows the editing of non-sourcecode text files. In this ”plain text” mode, code-related features such as coloring, case correction, auto complete are disabled. When saving plain text files, the IDE will not append its settings to the end of the file, even if this is configured for code files in the Preferences .
Whether or not a file is considered a code-file or not depends on the file extension. The standard PureBasic file extensions (pb, pbi and pbf) are recognized as code files. More file extensions can be recognized as code files by configuring their extension in the ”Editor” section of the Preferences .

Contents of the ”File” menu:



New
Create a new empty source code file.
Open
Open an existing source code file for editing.
Any text file will be loaded into the source-editing field. You can also load binary files with the Open menu. These will be displayed in the internal File Viewer .
Save

Saves the currently active source to disk. If the file isn't saved yet, you will be prompted for a filename. Otherwise the code will be saved in the file it was saved in before.

Save As...

Save the currently active source to a different location than it was saved before. This prompts you for a new filename and leaves the old file (if any) untouched.

Save All

Saves all currently opened sources.

Reload

Reloads the currently active source code from disk. This discards any changes not yet saved.

Close

Closes the currently active source code. If it was the only open code, the IDE will display a new empty file.

Close All

Closes all currently opened sources.

View changes

Shows the changes made to the current source code compared to its version that exists on the hard drive.

File format

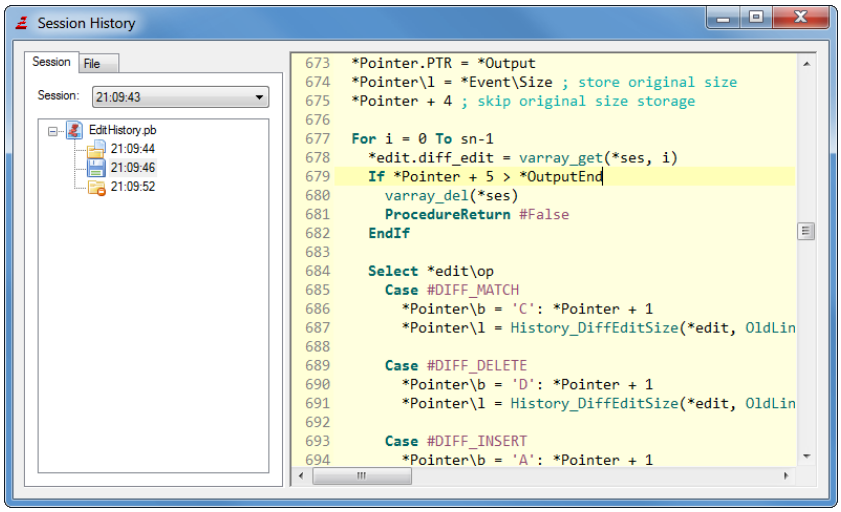
In this submenu you can select the text encoding as well as the newline format which should be used when the currently active source code is saved to disk. The IDE can handle files in Ascii or UTF-8. The newline formats it can handle are Windows (CRLF), Linux/Unix (LF) and MacOSX (CR). The defaults for newly created source codes can be set in the preferences .

Preferences

Here you can change all the settings that control the look & behavior of the IDE. For a detailed description of that see Customizing the IDE .

Session history

Session history is a powerful tool which regularly records changes made to any files in a database. A session is created when the IDE launch, and is closed when the IDE quits. This is useful to rollback to a previous version of a file, or to find back a deleted or corrupted file. It's like source backup tool, limited in time (by default one month of recording). It's not aimed to replace a real source code version control system like SVN or GIT. It's complementary to have finer change trace. The source code will be stored without encryption, so if you are working on sensitive source code, be sure to have this database file in a secure location, or disable this feature. To configure the session history tool, see preferences .



Recent Files

Here you can see a list of the last accessed files. Selecting a file in this submenu will open it again.

Quit

This of course closes the IDE. You will be asked to save any non-saved source codes.

Chapter 10

Editing features

The PureBasic IDE acts like any other Text Editor when it comes to the basic editing features. The cursor keys as well as Page Up/Page Down, Home and End keys can be used to navigate through the code. Ctrl+Home navigates to the beginning of the file and Ctrl+End to the End.
The default shortcuts Ctrl+C (copy), Ctrl+X (cut) and Ctrl+V (paste) can be used for editing. The "Insert" key controls whether text is inserted or overwritten. The Delete key does a forward delete.
Holding down the Shift key and using the arrow keys selects text.
The shortcuts Ctrl+Shift+Up and Ctrl+Shift+Down allow you to move a line up or down.
The shortcut Ctrl+D duplicates a selection.
In Windows, the shortcuts Ctrl+Plus and Ctrl+Minus allow you to zoom in and out the display of all tabs and Ctrl+0 restores the default zoom. Linux and OSX users can create them themselves. The shortcut Ctrl+mouse wheel only zooms the current tab.
The shortcuts Ctrl+Tab and Ctrl+Shift+Tab allow you to move around the open tabs.
There are many shortcuts available, like Ctrl+Shift+U/L/X to change the case, etc. Go to Preferences (menu "Files" / "Preferences / General / Shortcuts".
Furthermore, the IDE has many extra editing features specific to programming or PureBasic.

Indentation:

When you press enter, the indentation (number of space/tab at the beginning of the line) of the current and next line will be automatically corrected depending on the keywords that exist on these lines. A "block mode" is also available where the new line simply gets the same indentation as the previous one. The details of this feature can be customized in the preferences .

Tab characters:

By default, the IDE does not insert a real tab when pressing the Tab key, as many programmers see it as a bad thing to use real tabs in source code.
It instead inserts two spaces. This behavior can be changed in the Preferences. See Customizing the IDE for more information.

Special Tab behavior:

When the Tab key is pressed while nothing or only a few characters are selected, the Tab key acts as mentioned above (inserting a number of spaces, or a real tab if configured that way).
However when one or more full lines are selected, the reaction is different. In that case at the beginning of each selected line, it will insert spaces or a tab (depending on the configuration). This increases the indentation of the whole selected block.

Marking several lines of text and pressing Shift+Tab reverses this behavior. It removes spaces/tabs at the start of each line in order to reduce the indentation of the whole block.

Indentation/Alignment of comments:

Similar to the special tab behavior above, the keyboard shortcuts Ctrl+E and Ctrl+Shift+E (CMD+E and CMD+Shift+E on OSX) can be used to change the indentation of only the comments in a selected block of code. This helps in aligning comments at the end of code lines to make the code more readable. The used shortcut can be configured in the preferences .

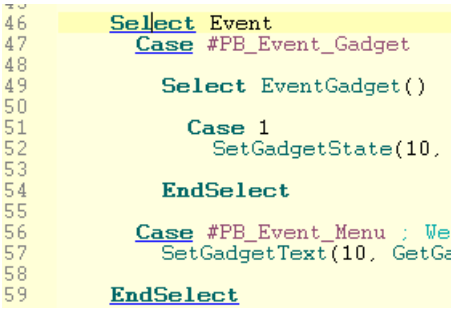
Selecting blocks of code:

The shortcut Ctrl+M (CMD+M on OSX) can be used to select the block of code that contains caret position (i.e. the surrounding If block, loop or procedure). Repeated usage of the shortcut selects further surrounding code blocks.
The shortcut Ctrl+Shift+M (CMD+Shift+M on OSX) reverses the behavior and reverts the selection to the block that was selected before the last usage of the Ctrl+M shortcut.
The used shortcuts can be configured in the preferences .

Double-clicking on source text:

Double-clicking on a word selects the whole word as usual. However in some cases, double-clicking has a special meaning:
When double-clicking on the name of a procedure call, while holding down the Ctrl Key, the cursor automatically jumps to the declaration of this procedure (the source containing this must currently open or part of the same project).
When double-clicking on an IncludeFile or XincludeFile statement, the IDE will try to open that file. (This is only possible if the included file is written as a literal string, and not through for example a constant.)
In the same way, if you double-click on an IncludeBinary statement, the IDE will try to display that file in the internal file viewer .

Marking of matching Braces and Keywords:

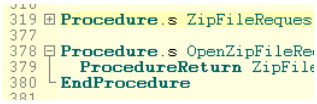


When the cursor is on an opening or closing brace the IDE will highlight the other brace that matches it. If a matching brace could not be found (which is a syntax error in PureBasic) the IDE will highlight the current brace in red. This same concept is applied to keywords. If the cursor is on a Keyword such as "If", the IDE will underline this keyword and all keywords that belong to it such as "Else" or "EndIf". If there is a mismatch in the keywords it will be underlined in red. The "Goto matching Keyword" menu entry described below can be used to quickly move between the matching keywords.
The brace and keyword matching can be configured in the Preferences .

Command help in the status bar:

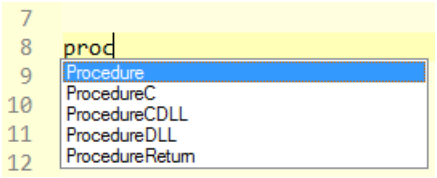
While typing, the IDE will show the needed parameters for any PureBasic function whose parameters you are currently typing. This makes it easy to see any more parameters you still have to add to this function. This also works for procedures , prototypes , interfaces or imported functions in your code as long as they are declared in the same source code or project .

Folding options:



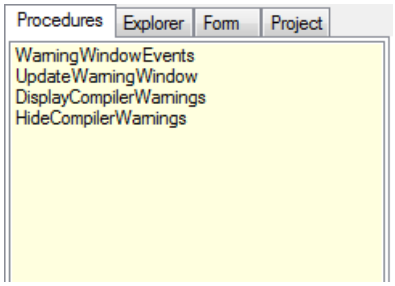
When special folding keywords are encountered (Procedure / EndProcedure by default. More can be added), the IDE marks the region between these keywords on the left side next to the line numbers with a [-] at the starting point, followed by a vertical line to the end point.
By clicking on the [-], you can hide ("fold") that section of source code to keep a better overview of larger source files. The [-] will turn into a [+]. By clicking again, the code will again be shown ("unfolded") again.
Note: Even though the state of these folded code lines is remembered when you save/reopen the file, the actual created code file always contains all lines. This only affects the display of the code in the IDE, not the code itself.
Another default fold keyword is ";{" and ";}". Since ";" marks a comment in PB, these will be totally ignored by the compiler. However, they provide the possibility to place custom fold points that do not correspond to a specific PB keyword.

Auto complete:



So that you do not have to remember the exact name of every command, there is the Auto complete feature to make things easier.
After you have typed the beginning of a command, a list of possible matches to the word start you have just typed will be displayed. A list of options is also displayed when you typed a structured variable or interface followed by a "\".
You can then select one of these words with the up/down keys and insert it at the point you are by pressing the Tab key. You can also continue typing while the list is open. It will select the first match that is still possible after what you typed, and close automatically when either you have just typed an exact match or if there are no more possible matches in the list.
Escape closes the auto complete list at any time. It also closes if you click with the mouse anywhere within the IDE.
Note: You can configure what is displayed in the Auto complete list, as well as turning off the automatic popup (requiring a keyboard shortcut such as Ctrl+Space to open list) in the Preferences. See the Auto complete section of Customizing the IDE for more information.

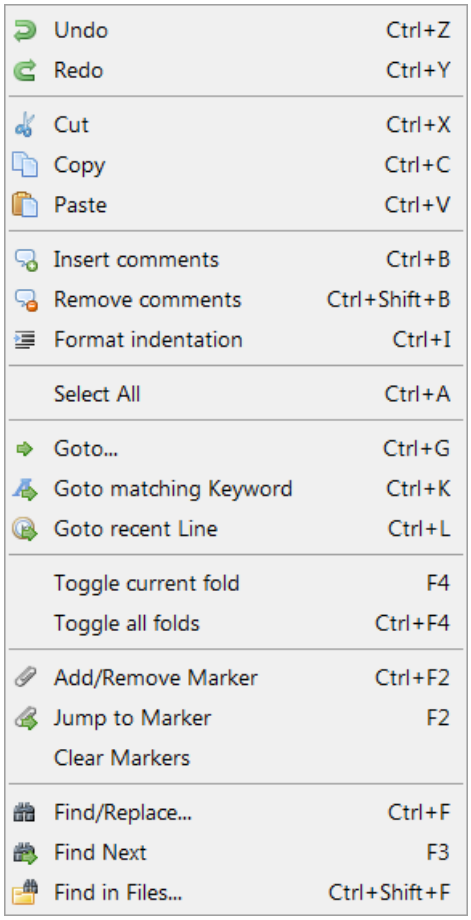
Tools Panel on the side:



Many tools to make navigating/editing the source code easier can be added to the Tools Panel on the side of the editor window. For an overview of them and how to configure them, see Built-in Tools .

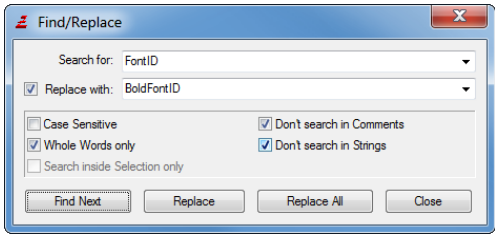
The Edit Menu:

Following is an explanation of the Items in the Edit menu. Note that many of the Edit menu items are also accessible by right clicking on the source code, which opens a popup menu.



Undo
Undoes the last done action in the code editing area. There is an undo buffer, so several actions can be undone.
Redo
Redo the last action undone by the undo function.
Cut
Copy the selected part of the source code to the clipboard and remove it from the code.
Copy
Copy the selected text to the Clipboard without deleting it from the code.

Paste
Insert the content of the Clipboard at the current position in the code. If any text is selected before this, it will be removed and replaced with the content of the Clipboard.
Insert comments
Inserts a comment (";") before every line of the selected code block. This makes commenting large blocks of code easier than putting the ; before each line manually.
Remove comments
Removes the comment characters at the beginning of each selected line. This reverts the "Insert comments" command, but also works on comments manually set.
Format indentation
Reformats the indentation of the selected lines to align with the code above them and to reflect the keywords that they contain. The rules for the indentation can be specified in the preferences .
Select all Selects the whole source code.
Goto
This lets you jump to a specific line in your source code.
Goto matching Keyword
If the cursor is currently on a keyword such as "If" this menu option jumps directly to the keyword that matches it (in this case "EndIf").
Goto recent line
The IDE keeps track of the lines you view. For example if you switch to a different line with the above Goto function, or with the Procedure Browser tool. With this menu option you can jump back to the previous position. 20 such past cursor positions are remembered.
Note that this only records greater jumps in the code. Not if you just move up/down a few lines with the cursor keys.
Toggle current fold
This opens/closes the fold point in which the cursor is currently located.
Toggle all Folds
This opens/closes all fold points in the current source. Very useful to for example hide all procedures in the code. Or to quickly see the whole code again when some of the code is folded.
Add/Remove Marker
Markers act like Bookmarks in the source code. There presence is indicated by a little arrow next to the line numbers. You can later jump to these markers with the "Jump to marker" command.
The "Add/Remove Marker" sets or removes a marker from the current line you are editing.
Note: You can also set/remove markers by holding down the Ctrl Key and clicking on the border that holds the markers (not the Line-number part of it).
Jump to Marker
This makes the cursor jump to the next marker position further down the code from the current cursor position. If there is no marker after the cursor position, it jumps to the first on in the source code. So by pressing the "Jump to Marker" shortcut (F2 by default) several times, you can jump to all the markers in the code.
Clear Markers This removes all markers from the current source code.
Find/Replace



The find/replace dialog enables you to search for specific words in your code, and also to replace them with something else.
The "Find Next" button starts the search. The search can be continued after a match is found with the Find Next menu command (F3 by default).
You can make the search more specific by enabling one of the checkboxes:
Case Sensitive : Only text that matches the exact case of the search word will be found.
Whole Words only : Search for the given word as a whole word. Do not display results where the search word is part of another word.

Don't search in Comments : Any match that is found inside a comment is ignored.
Don't search in Strings : Any match that is found inside a literal string (in " ") is ignored.
Search inside Selection only : Searches only the selected region of code. This is really useful only together with the "Replace All" button, in which case it will replace any found match, but only inside the selected region.
By enabling the "Replace with" checkbox, you go into replace mode. "Find Next" will still only search, but with each click on the "Replace" button, the next match of the search word will be replaced by whatever is inside the "Replace with" box.
By clicking on "Replace All", all matches from the current position downwards will be replaced (unless "Search inside Selection only" is set).

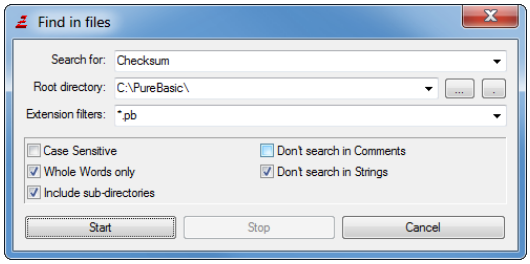
Find Next

This continues the search for the next match of the last search started by the Find/Replace dialog.

Find Previous

This continues the search for the previous match of the last search started by the Find/Replace dialog.

Find in Files



The Find in Files Dialog lets you carry out a search inside many files in a specific directory.
You have to specify a search keyword, as well as a base directory ("root directory") in which to search. You can customize the searched files by specifying extension filters. Any number of filters can be given separated by ",". (*.* or an empty extension field searches all files). As with "Find/Replace", there are checkboxes to make the search more specific.
The "Include sub-directories" checkbox makes it search (recursively) inside any subdirectory of the given root directory too.
When starting the search, a separate window will be opened displaying the search results, giving the file, line number as well as the matched line of each result.
Double-clicking on an entry in the result window opens that file in the IDE and jumps to the selected result line.

Chapter 11

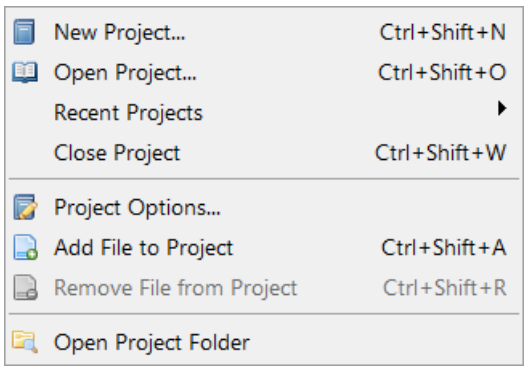
Managing projects

The IDE comes with features to easily handle larger projects. These features are completely optional. Programs can be created and compiled without making use of the project management. However, once a program consists of a number of source code and maybe other related files, it can be simpler to handle them all in one project.

Project management overview

A project allows the management of multiple source codes and other related files in one place with quick access to the files through the project tool . Source files included in a project can be scanned for AutoComplete even if they are not currently open in the IDE. This way functions, constants, variables etc. from the entire project can be used with AutoComplete. The project can also remember the source files that are open when the project is closed and reopen them the next time to continue working exactly where you left off.
Furthermore, a project keeps all the compiler settings in one place (the project file) and even allows to manage multiple "compile targets" per project. A compile target is just a set of compiler options. This way multiple versions of the same program, or multiple smaller programs in one project can be easily compiled at once.
To compile a project from a script or makefile, the IDE provides command-line options to compile a project without opening a user interface. See the section on command-line options for more details.
All filenames and paths in a project are stored relative to the project file which allows a project to be easily moved to another location as long as the relative directory structure remains intact.

The Project menu

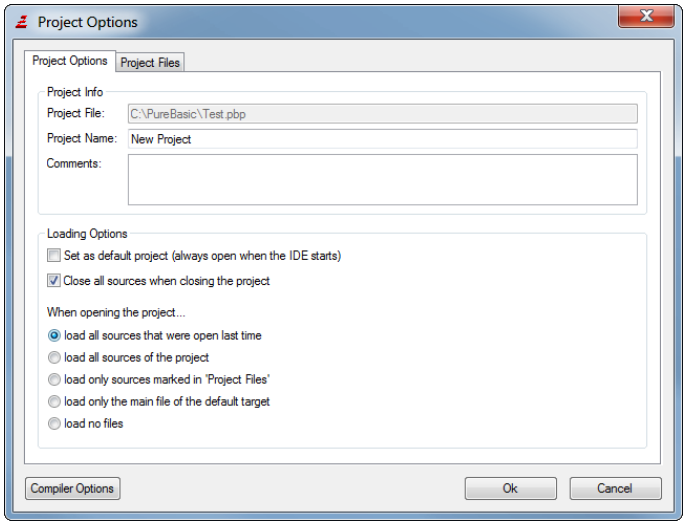


New Project
Creates a new project. If there is a project open at the time it will be closed. The project options window will be opened where the project filename has to be specified and the project can be configured.

Open Project
Opens an existing project. If there is a project open at the time it will be closed. Previously open source codes of the project will be opened as well, depending on the project configuration.
Recent Projects
This submenu shows a list of recently opened project files. Selecting one of the entries opens this project.
Close Project
Closes the currently open project. The settings will be saved and the currently open source files of the project will be closed, depending on the project configuration.
Project Options
Opens the project options window. See below for more information.
Add File to Project
Adds the currently active source code to the current project. Files belonging to the project are marked with a ”>” in the file panel.
Remove File from Project
Removes the currently active source from the current project.
Open Project folder
Opens the folder that contains the project file in whatever file manager is available on the system.

The project options window

The project options window is the central configuration for the project. The general project settings as well as the settings for the individual files in the project can me made here.

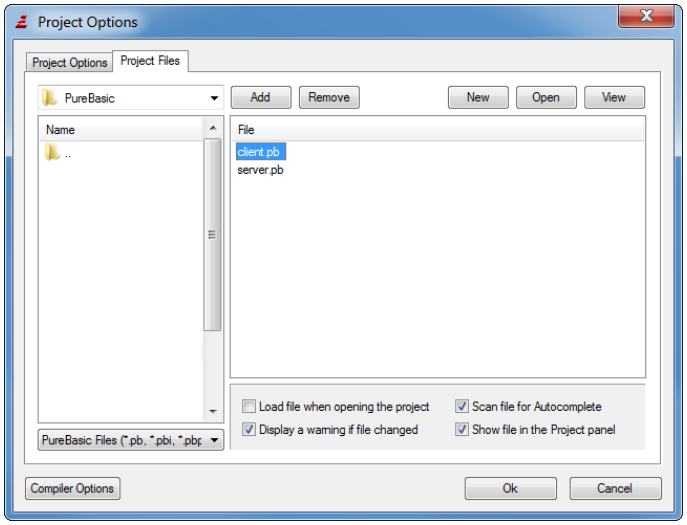


The following settings can be made on the ”Project Options” tab:
Project File
Shows the filename of the project file. This can only be changed during project creation.
Project Name
The name of the project. This name is displayed in the IDE title bar and in the ”Recent Projects” menu.
Comments
This field allows to add some comments to the project. They will be displayed in the project info tab.
Set as default project
The default project will be loaded on every start of the IDE. Only one project can be the default project at a time. If there is no default project, the IDE will load the project that was open when the IDE was closed last time if there was one.
Close all sources when closing the project
If enabled, all sources that belong to the project will be closed automatically when the project is closed.
When opening the project...

> **load all sources that where open last time**
> When the project is opened, all the sources that were open when the project was closed will be opened again.

load all sources of the project
When the project is opened, all (source-)files of the project will be opened.
load only sources marked in 'Project Files'
When the project is opened, only the files that are marked in the 'Project Files' tab will be opened. This way you can start a session always with this set of files open.
load only the main file of the default target
When the project is opened, the main file of the default target will be opened too.
load no files
No source files are opened when the project is opened.

The "Project Files" tabs shows the list of files in the project on the right and allows changing their settings. The explorer on the left is for the selection of new files to be added.



The buttons on the top have the following function:

Add
Add the selected file(s) in the explorer to the project.
Remove
Remove the selected files in the file list from the project.
New
Shows a file requester to select a filename for a new source file to create. The new file will be created, opened in the IDE and also added to the project.
Open
Shows a file requester to select an existing file to open. The file will be opened in the IDE and added to the project.
View
Opens the selected file(s) in the file list in the IDE or if they are binary files in the FileViewer.

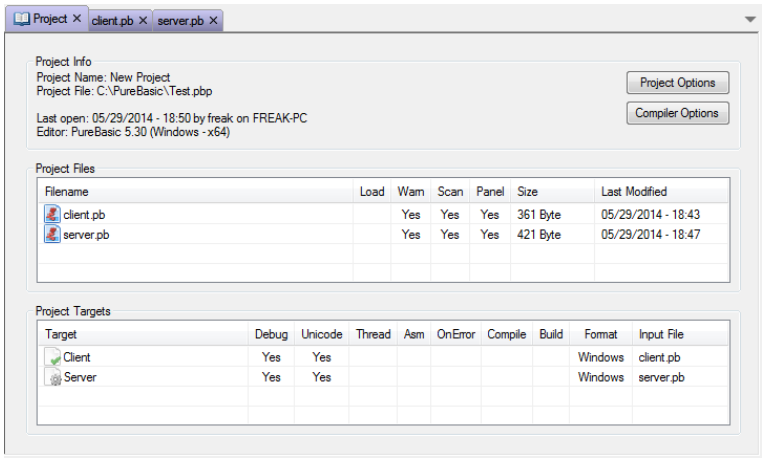
The checkboxes on the bottom specify the options for the files in the project. They can be applied to a single file or to multiple files at once by selecting the files and changing the state of the checkboxes. The settings have the following meaning:

Load file when opening the project
Files with this option will be loaded when the project is open and the "load only sources marked in 'Project Files"' option is specified on the "Project Options" tab.
Display a warning if file changed
When the project is closed, the IDE will calculate a checksum of all files that have this option set and display a warning if the file has been modified when the project is opened the next time. This allows to be notified when a file that is shared between multiple projects has been edited while working on another project. This option should be disabled for large data files to speed up project loading and saving, or for files which are changed frequently to avoid getting a warning every time the project is opened.

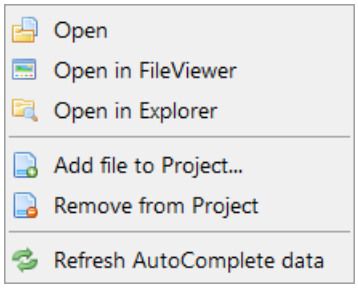
Scan file for AutoComplete
Files with this option will be scanned for AutoComplete data even when they are not currently loaded in the IDE. This option is on by default for all non-binary files. It should be turned off for all files that do not contain source code as well as for any files where you do not want the items to turn up in the AutoComplete list.
Show file in Project panel
Files with this option will be displayed in the project side-panel. If the project has many files it may make sense to hide some of them from the panel to have a better overview and faster access to the important files in the project.

The project overview

When a project is open, the first tab of the file panel shows an overview of the project and its files.



Project Info
This section shows some general info about the project, such as the project filename, its comments or when and where the project was last opened.
Project Files
This section shows all files in the project and their settings from the Project Options window. Double-clicking on one of the files opens the file in the IDE. Right-clicking displays a context menu with further options:



Open - Open the file in the IDE.
Open in FileViewer - Open the file in the FileViewer of the IDE.
Open in Explorer - Open the file in the operating systems file manager.
Add File to Project - Add a new file to the project.
Remove File from Project - Remove the selected file(s) from the project.
Refresh AutoComplete data - Rescan the file for AutoComplete items.

Project Targets
This section shows all compile targets in the project and some of their settings. Double-clicking on one of the targets opens this target in the compiler options . Right-clicking on one of the targets displays a context menu with further options:

Edit target - Open the target in the compiler options.
Set as default target - Set this target as the default target.
Enable in 'Build all Targets' - Include this target in the 'Build all Targets' compiler menu option.

The project panel

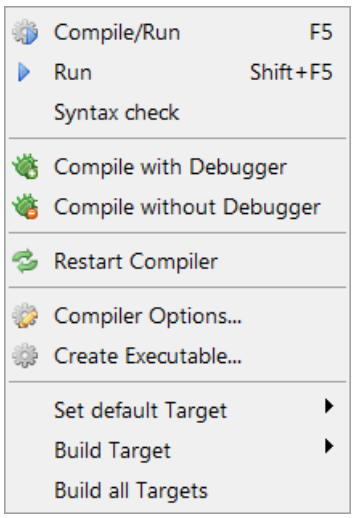
There is a sidepanel tool which allows quick access to the files belonging to the project. For more information see the built-in tools section.

Chapter 12

Compiling your programs

Compiling is simple. Just select ”Compile/Run” (F5 by default) and your program will be compiled and executed for a testing run.
To customize the compiling process, you can open the ”Compiler options” dialog. The settings made there are associated with the current source file or the current project, and also remembered when they are closed. The place where this information is saved can be configured. By default, it is saved at the end of the source code as a comment (invisible in the IDE).
In case of an error that prevents the compiler from completing the compilation, it aborts and displays an error-message. This message is also logged in the error log, and the line that caused the error is marked.
A number of functions from older versions of PureBasic that have been removed from the package still exist for a while as a compatibility wrapper to allow older codes to be tested/ported more easily. If such a function is used in the code, the compiler will issue a warning. A window will be opened displaying all warnings issued during compilation. Double-clicking on a warning will display the file/line that caused the warning. Note that such compatibility wrappers will not remain indefinitely but will be removed in a future update, so it is recommended to fix issues that cause a compiler warning instead of relying on such deprecated functions.

The compiler menu



Compile/Run
This compiles the current source code with the compiler options set for it and executes it. The executable file is stored in a temporary location, but it will be executed with the current path set to the directory of the source code; so loading a file from the same directory as the source code will work.
The source code need not be saved for this (but any included files must be saved).
The ”Compile/Run” option respects the debugger setting (on or off) from the compiler options or

debugger menu (they are the same).

Run

This executes the last compiled source code once again. Whether or not the debugger is enabled depends on the setting of the last compilation.

Compile with Debugger

This is the same as "Compile/Run" except that it ignores the debugger setting and enabled the debugger for this compilation. This is useful when you usually have the debugger off, but want to have it on for just this one compilation.

Compile without Debugger

Same as "Compile with Debugger" except that it forces the debugger to be off for this compilation.

Restart Compiler (not present on all OS)

This causes the compiler to restart. It also causes the compiler to reload all the libraries and resident files, and with that, the list of known PureBasic functions, Structures, Interfaces and Constants is updated too. This function is useful when you have added a new User Library to the PB directory, but do not want to restart the whole IDE. It is especially useful for library developers to test their library.

Compiler Options

This opens the compiler options dialog, that lets you set the options for the compilation of this source file.

Create executable

This opens a save dialog, asking for the executable name to create. If the executable format is set to DLL, it will create a DLL on Windows, shared object on Linux and dylib on OS X. When creating an executable on OS X, appending '.app' at the executable name will create a bundled executable with the necessary directory structure, including the icon. If no '.app' is set, then it will create a regular console-like executable.

Set default Target

When a project is open, this submenu shows all compile targets and allows to quickly switch the current default target. The default target is the one which is compiled/executed with the "Compile/Run" menu entry.

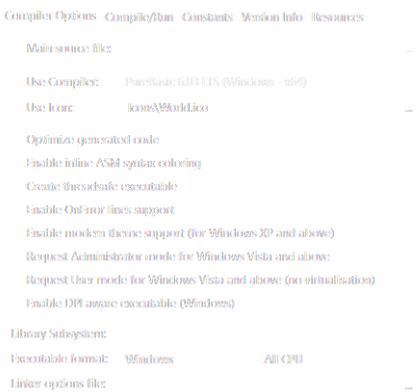
Build Target

When a project is open, this submenu shows all compile targets and allows to directly compile one of them.

Build all Targets

When a project is open, this menu entry compiles all targets that have this option enabled in the compiler options. A window is opened to show the build progress.

Compiler options for non-project files



Main source file

By enabling this option, you can define another file that will be the one sent to the compiler instead of this one. The use of this is that when you are editing a file that does not run by itself, but is included into another file, you can tell the compiler to use that other file to start the compilation.

Note: When using this option, you MUST save your source before compiling, as only files that are written to disk will be used in this case. Most of the compiler settings will be taken from the main source file, so when setting this, they are disabled. Only some settings like the debugger setting will be

used from the current source.

Use Compiler

This option allows the selection of a different compiler to use instead of the compiler of the current PureBasic version. This makes it easy to compile different versions of the same program (x86 and x64) without having to start up the IDE for the other compiler just for the compilation. Additional compilers for this option have to be configured in the preferences .

If the compiler version matches that of the default compiler but the target processor is different then the built-in debugger of the IDE can still be used to debug the compiled executable. This means that an executable compiled with the x86 compiler can be debugged using the x64 IDE and vice versa. If the version does not match then the standalone debugger that comes with the selected compiler will be used for debugging to avoid version conflicts.

Use Icon (Windows and MacOS X only)

Here you can set an icon that will be displayed when viewing the created executable in the explorer. It is also displayed in the title bar of your programs windows and the Taskbar.

Windows: The icon must be in ICO format (Windows Icon).

MacOS X: The icon must be in ICNS format (Macintosh Icon). To create such an icon, you should create PNG files in the dimensions 128x128, 48x48, 32x32 and 16x16 of your image and then use the tool "Icon Composer" that comes with the OSX developer tools to create the ICNS file. It should be located in /Developer/Applications/Utilities/. To be displayed once the application has just been created, the Finder may need to be restarted.

Optimize generated code

This enables the C code optimizer. It simply sets gcc's optimization flags on -O2 when enabled, on -O0 otherwise. See here.

Enable inline ASM syntax coloring

This enables the inline ASM syntax coloring. See the Inline x86 ASM section of the help-file for more information on this option.

Create thread-safe executable

This tells the compiler to use a special version of certain commands to make them safe to be used in threads. See the Thread library for more information.

This also enables the Debugger to display correct information if threads are used. Without this option, the debugger might output wrong line numbers when threads are involved for example.

Enable modern theme support (Windows only)

Adds support for skinned windows on Windows Vista, Windows 7 or Windows 8.

Request Administrator mode for Windows Vista and above (Windows only)

The created executable will always be started with administrator rights on Windows Vista and above (it will not launch if the administrator password is not entered). This option should be set for programs that need to access restricted folders or restricted areas of the registry to get full access.

If this option is turned on, the standalone debugger will automatically selected when debugging, so the program can be tested in administrator mode.

Note: This option has no effect when the program is run on other versions of Windows.

Request User mode for Windows Vista and above (Windows only)

This option disables the "Virtualization" feature for this executable on Windows Vista and above. Virtualization caused file and registry access to be redirected to a special user folder if the user does not have the needed rights to do the operation (this is done for compatibility with older programs).

Note that this redirection is done without notifying the user; this can lead to some confusion if he tries to find saved files on the file-system. Because of this, it is recommended to disable this feature if the program complies with the Windows Vista file/registry access rules.

Note: This option has no effect when the program is run on other versions of Windows. It cannot be combined with the "Administrator mode" option above.

Enable DPI Aware Executable (Windows only)

This option enable DPI awareness when creating an executable. That means than GUI created in PureBasic will scale automatically if the DPI of the screen is above 100%. Most of the process is seamless, but some case needs to be worked out, like pixel based gadgets (ImageGadget, CanvasGadget etc.).

Enable DLL Protection (Windows only)

Enable DLL preloading protection to the executable. It prevents that system DLLs are first searched for in the program directory instead of in the System32 directory of the Windows operating system.

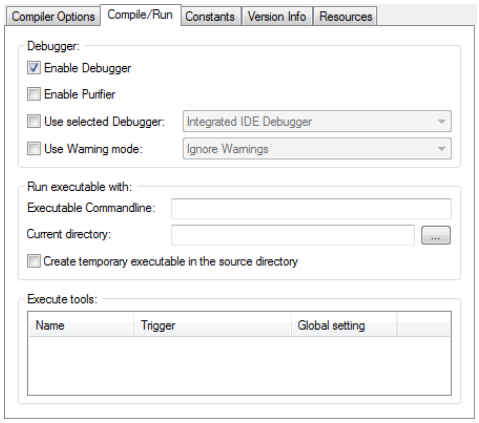
Enable OnError lines support (Windows only)

Includes line numbers information with the executable for the OnError-Library .

Library Subsystem
Here you can select different subsystems for compilation. More than one subsystem can be specified, separated with a comma. For more information, see subsystems .
Executable format
This allows you to specify the created executable format:
Windows : a normal windows executable.
Console : an executable with a default console. This one still can create windows and such, but it always has a console open. When executed from a command prompt, this executable type uses the command terminal as its console and writes there, whereas the "Windows" executable would create a separate Console window when using OpenConsole() . This setting must be used to create a Console application that can have its input/output redirected with pipes.
Shared DLL : create a windows DLL. See Building a DLL for more info.
Note: When you do "Compile/Run" with a DLL source code, it is executed as a normal executable. A dll is only created when you use "create executable".
Cpu Optimisation (next to Executable format)
This setting allows to include Cpu optimised PB functions in your executable:
All CPU : The generic functions are included that run on all CPUs.
Dynamic CPU : The generic functions as well as any available CPU specific function are included. The function to execute is decided at runtime. This creates a bigger executable, but it will run as fast as possible on all CPUs.
All other options : Include only the functions for a specific CPU. The executable will not run on any Cpu that does not support this feature.
Note: No PB functions actually support this feature for now (it is ignored for them). However, some User Libraries include such optimisations.
Linker options file
A textfile can be specified here with further command-line options that should be passed to the linker when creating the executable. The file should contain one option per line.

Compile/Run

This section contains options that affect how the executable is run from the IDE for testing. Except for the tools option, they have no effect when the "Create executable" menu is used.

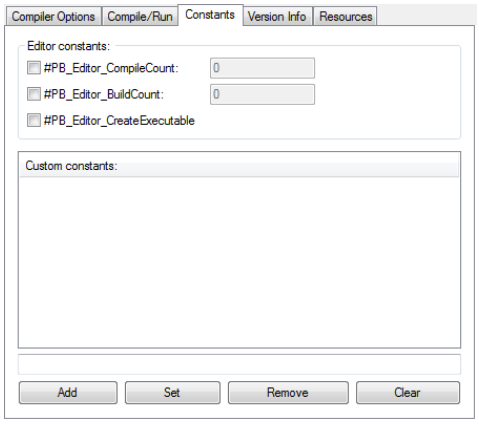


Enable Debugger
This sets the debugger state (on/off) for this source code, or if the main file option is used, for that file too. This can also be set from the debugger menu.
Enable Purifier
This enables purifier support for the debugger. The purifier can detect a certain type of programming errors such as writing past the end of an allocated memory buffer. See Included debugging tools for more details.
Use selected Debugger
This allows to choose a different debugger type for this file only. If this option is disabled, the default debugger is used; this can be specified in the preferences .
Use Warning mode

This allows to choose a different warning mode for this file only. If this option is disabled, the default setting is used which can be specified in the preferences . The available options are:
Ignore Warnings: Warnings will be ignored without displaying anything.
Display Warnings: Warnings will be displayed in the error log and the source code line will be marked, but the program continues to run.
Treat Warnings as Errors: A warning will be treated like an error.
Executable command-line
The string given here will be passed as the command-line to the program when running it from the IDE. The content of this string can be got with ProgramParameter() .
Current directory
The directory specified here will be set as the current directory for the program when running it from the IDE.
Create temporary executable in the source directory
With this option turned on, the temporary executable file for running the program from the IDE will be placed inside the source directory. This can be useful if the program depends on files inside the source directory for testing. With this option turned off, the executable is created in the systems temporary directory.
Execute tools
Here external tools can be enabled on a per-source basis. The "Global settings" column shows if the tool is enabled or disabled in the tools configuration . A tool will only be executed for the source if it is both enabled globally and for this source.
Note: For a tool to be listed here, it must have the "Enable Tool on a per-source basis" option checked in the tools configuration and be executed by a trigger that is associated with a source file (i.e. not executed by menu or by editor startup for example).

Constants

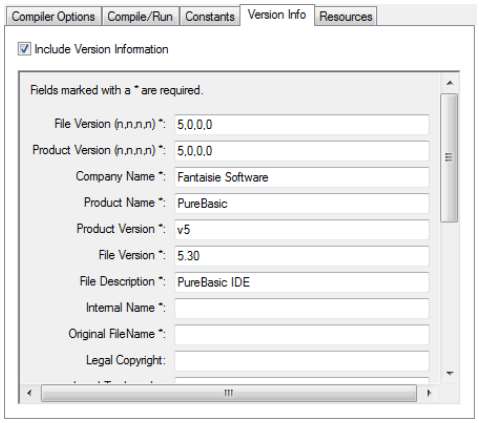
In this section, a set of special editor constants as well as custom constants can be defined which will be predefined when compiling this source.



#PB_Editor_CompileCount
If enabled, this constant holds the number of times that the code was compiled (both with "Compile/Run" and "Create Executable") from the IDE. The counter can be manually edited in the input field.
#PB_Editor_BuildCount
If enabled, this constant holds the number of times that the code was compiled with "Create Executable" only. The counter can be manually edited in the input field.
#PB_Editor_CreateExecutable
If enabled, this constant holds a value of 1 if the code is compiled with the "Create Executable" menu or 0 if "Compile/Run" was used.
Custom constants
Here, custom constants can be defined and then easily switched on/off through checkboxes. Constant definitions should be added as they would be written within the source code. This provides a way to enable/disable certain features in a program by defining a constant here and then checking in the source

for it to enable/disable the features with CompilerIf/CompilerEndIf .
Inside the definition of these constants, environment variables can be used by specifying them in a "bash" like style with a "$" in front. The environment variable will be replaced in the constant definition before compiling the source. This allows to pass certain options of the system that the code is compiled on to the program in the form of constants.
Example: #Creator=$USERNAME
Here, the $USERNAME will be replaced by the username of the logged in user on Windows systems. If an environment variable does not exist, it will be replaced by an empty string.
Note: To test within the source code if a constant is defined or not, the Defined() compiler function can be used.

Version Information

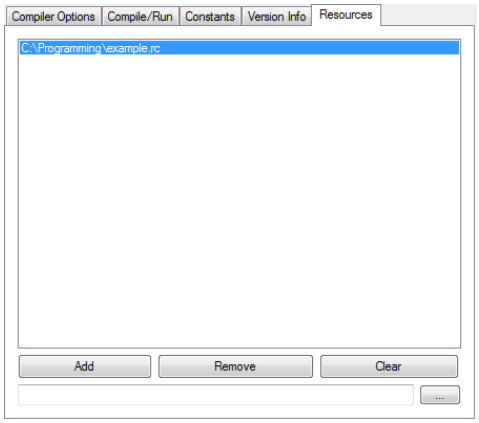


This is a Windows only feature. By enabling this, a resource is included in the executable with information about your program. It can be viewed by right-clicking on the executable in the windows explorer and selecting "Properties". Also it can be read by other programs such as setup tools.
Fields marked with a * are required if you want to include the version info (if not all required fields are set, the information may not display correctly on some versions of Windows).
The first two fields MUST be composed of 4 numbers separated by commas. All other fields may be any string. In the 3 empty boxes, you can define your own fields to include in the Version info block.
In all the string fields, you may include special tokens that are replaced when compiling:
%OS : replaced with the version of Windows used to compile the program
%SOURCE : replaced with the filename (no path) of the source file.
%EXECUTABLE : replaced with the name of the created executable (this only works when "create executable" is used, not with "Compile/Run".
%COMPILECOUNT : replaced with the value of the #PB_Editor_CompileCount constant.
%BUILDCOUNT : replaced with the value of the #PB_Editor_BuildCount constant.
Furthermore, you can use any token listed with the FormatDate() command. These tokens will be replaced with their respective meaning in FormatDate() used with the date of the compilation (i.e. %yy gives the year of the compilation)
Meaning of the lower 3 fields:
File OS
Specifies the OS that this Program is compiled for (Using VOS_DOS or VOS_WINDOWS16 makes little sense. They are only included for the sake of completeness).
File Type
Type of the executable (Here VFT_UNKNOWN, VFT_APP or VFT_DLL are the only ones that really make sense for PureBasic programs).
Language
Specifies the language in which this version info is written.
The fields values can be accessed when compiling the program from the IDE using the following constants (same order):

```
#PB_Editor_FileVersionNumeric
#PB_Editor_ProductVersionNumeric
#PB_Editor_CompanyName
```

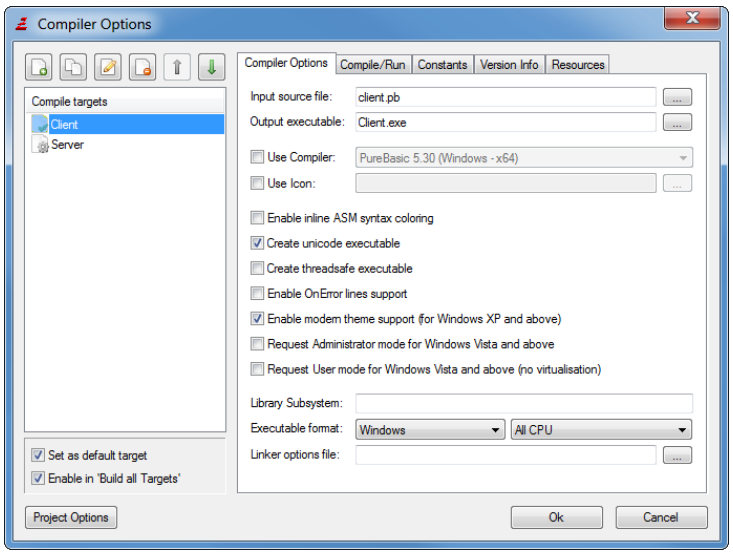
```
#PB_Editor_ProductName
#PB_Editor_ProductVersion
#PB_Editor_FileVersion
#PB_Editor_FileDescription
#PB_Editor_InternalName
#PB_Editor_OriginalFilename
#PB_Editor_LegalCopyright
#PB_Editor_LegalTrademarks
#PB_Editor_PrivateBuild
#PB_Editor_SpecialBuild
#PB_Editor_Email
#PB_Editor_Website
```

Resources



This is a Windows only feature. Here you can include as many Resource scripts (*.rc files) as you want. They will be compiled and included with the executable. You can use any resource editor (for example the PellesC IDE) to create such scripts.
Note: Since Resources are a specific to the Windows platform only, PB does not include a Library to manage them and they are not further documented here. See documentation on the Windows API and resources for more information.

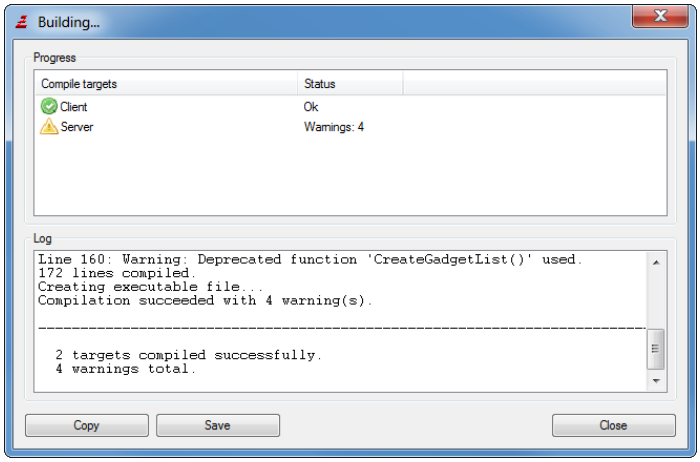
Compiler options for projects



The compiler options for projects allow the definition of multiple compile targets. Each target is

basically a set of compiler options with a designated source file and output executable. The left side of the compiler options window is extended with the list of the defined compile targets. The toolbar on top of it allows to create, delete, copy, edit or move targets in the list.
The default target is the one which will be compiled when the ”Compile/Run” menu entry is selected. It can be quickly switched with the ”Set as default target” checkbox or from the compiler menu. The ”Enable in 'Build all Targets”' option specifies whether or not the selected target will be built when the 'Build all Targets' menu entry is used.
The right side of the compiler options is almost the same as in the non-project mode and reflects the settings for the compile target that is currently selected on the left. The only difference are the ”Input source file” and ”Output executable” fields on the first tab. These fields have to be specified for all compile targets. Other than that, the compiler options are identical to the options described above.
In project mode, the information about the compile target is stored in the project file and not in the individual source files. Information that belongs to the file (such as the folding state) are still saved for the individual source files in the location specified by the Preferences .

The Build progress window



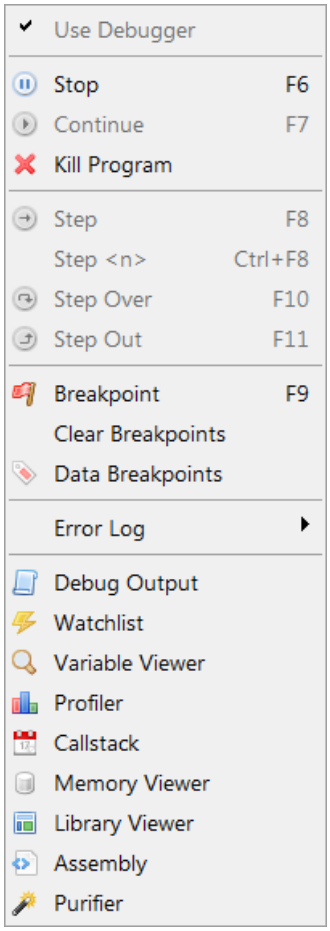
When the 'Build all Targets' menu entry is selected on an open project, all targets that have the corresponding option set in the compiler options will be compiled in the order they are defined in the compiler options. The progress window shows the current compile progress as well as the status of each target. When the process is finished, the build log can be copied to the clipboard or saved to disk.

Chapter 13

Using the debugger

PureBasic provides a powerful debugger that helps you find mistakes and bugs in your source code. It lets you control the program execution, watch your variables , arrays or lists or display debug output of your programs. It also provides advanced features for assembly programmer to examine and modify the CPU registers or view the program stack, or the Memory of your program. It also provides the possibility to debug a program remotely over the network.
To enable the debugger for your program, you can select "Use Debugger" from the debugger menu, or set it in your programs Compiler options. By using the "Compile with Debugger" command from the Compiler menu, you can enable the debugger for just one compilation.
You can directly use debugger commands in your source, such as CallDebugger, Debug, DebugLevel, DisableDebugger and EnableDebugger.
The PureBasic debugger comes in 3 forms:
A Debugger integrated directly with the IDE, for an easy to use, quick way to debug your programs directly from the programming environment. This debugger also provides the most features.
A separate, standalone debugger, that is useful for some special purposes (for example, when the same program must be executed and debugged several times at once) or to be used with third party code Editors. It provides most of the features of the integrated IDE debugger, but because it is separate from the IDE, some of the efficiency of the direct access from the IDE is lost. The standalone debugger can be used to debug programs remotely through a network connection.
A console only debugger. This debuggers primary use is for testing non-graphical environment like on Linux systems without an X server, or to remotely develop through ssh.
The type of debugger that is used can be selected in the preferences .
All this debugging functionality however comes at a price. Running a program in debug mode is significantly slower in its execution that running it without the debugger. This should be no problem however, since this is for testing only anyway.
If you need to use the debugger, but have some parts in you program that require the full execution speed, you can disable the debugger in just that section with the DisableDebugger / EnableDebugger keywords.

The Debugger integrated into the IDE



You can access all the debugger features while the program is running from the debugger menu, or the corresponding toolbar buttons or shortcuts.
While you are debugging your program, all the source files that belong to that program (also included files) will be locked to read-only until the program has finished. This helps to ensure that the code that is marked as the currently executed one is has not actually been modified already without a recompilation.
Note that a program can be run only one time at once in IDE debugger mode. If you try to executed it again, you are given the option to execute it with the standalone Debugger.
Tip:
The debugger menu is also added to the system-menu of the Main IDE window (the menu you get when clicking the PB icon in the left/top of the window). This allows you to access the debugger menu also from the Taskbar, by right-clicking on the Taskbar-Icon of the IDE.

Program Control

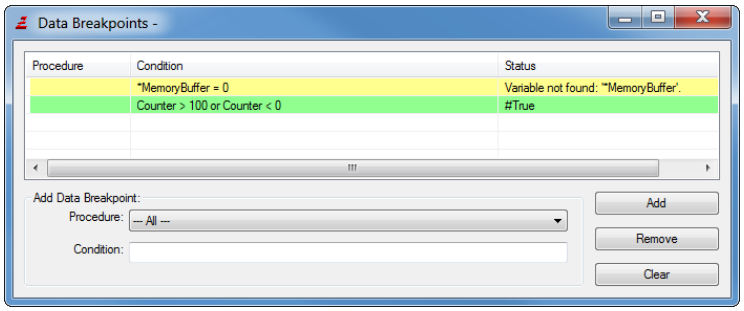
There are functions for basic control of the running program. You can halt the execution to examine variables and the code position or let the code execute line by line to follow the program flow. While the program is halted, the line that is currently being executed is marked in your source code (with very light-blue background color in the default colors).
The state of the program can be viewed in the IDE status bar, and in the Error log area.
Menu commands for program control:
Stop
Halts the Program and displays the current line.
Continue
Continues the program execution until another stop condition is met.
Kill Program
This forces the program to end, and closes all associated debugger windows.

Step
This executes one line of source code and then stops the execution again.
Step <n>
This will execute a number of steps that you can specify and then stop the execution again.
Step Over
This will execute the current line in the source and then stop again, just like the normal 'Step'. The difference is that if the current line contains calls to procedures , the execution will not stop inside these procedures like it does with the normal 'Step', but it will execute the whole procedure and stop after it returned. This allows to quickly skip procedures in step mode.
Step Out
This will execute the remaining code inside the current procedure and stop again after the procedure has returned. If the current line is not in any procedure, a normal 'Step' will be done.

Line Breakpoints

Breakpoints are another way to control the execution of your program. With the Breakpoint menu command, you mark the currently selected line as a breakpoint (or remove any breakpoint that exists in that line).
When the execution of the code reaches that line, it will stop at this point. Note that if you select a non-executable line (such as an empty line or a Structure definition), it will halt the execution on the next executable line after that.
After the execution of your program has stopped at a breakpoint, you can use any of the Program control commands to continue/end the execution.
Breakpoints can be set and removed dynamically, while your program is running, or while you are editing your source code. With the "Clear Breakpoints" command, you can clear all breakpoints in a source file.
Note: You can also set/remove Breakpoints by holding down the Alt Key and clicking on the border that contains the Breakpoint marks.

Data Breakpoints



In addition to the line specific breakpoints, the debugger also provides data breakpoints. Data breakpoints halt the program if a given condition is met. This way it is easy to find out when a variable or other value in the program changes and halt the program if that happens. The condition can be any PureBasic expression that can be evaluated to true or false. This can be anything that could be put after an If keyword, including logical operators such as And, Or or Not. Most functions of the Math , Memory and String libraries as well as all object validation functions in the form IsXXX() and the XxxID functions for getting the OS identifiers for an object are also available.
Example conditions:

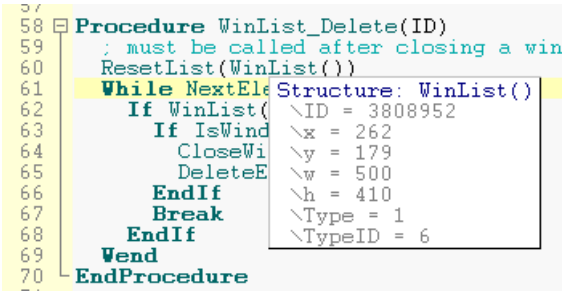
```
1  MyVariable$ <> "Hello" Or Counter < 0  ; halt if MyVariable$ changes
    from "Hello" or if the Counter falls below zero
2  PeekL(*SomeAddress+500) <> 0           ; halt if the long value at
    the given memory location is not equal to zero
```

Data breakpoints can be added using the Data Breakpoint option from the Debugger menu. They can be limited to a specific procedure or they can be added for all code. The special "Main" entry of the

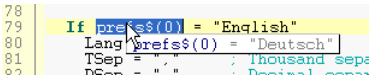
procedure selection specifies that the data breakpoint should only be checked when the execution is not in any procedure.
The status column shows the status of all breakpoint conditions on their last evaluation. This can be true, false or an error if the condition is not a valid expression. Once a condition is evaluated to true, the program execution will be halted. This condition is automatically removed from the list as soon as the program continues, so that it does not halt the program again immediately.
Note: Checking for data breakpoints slows down the program execution because the breakpoint conditions have to be re-evaluated for every executed line of code to check if the condition is met. So data breakpoints should only be added when needed to keep the program execution fast otherwise. Limiting a data breakpoint to a certain procedure also increases the speed because the check then only affects the given procedure and not the entire program.

Examining variables during runtime

The value of a variable can be very quickly viewed while the program is running by placing the mouse cursor over a variable in the source code and waiting for a brief moment. If the variable is currently in scope and can be displayed, its value will be shown as a tool-tip on the mouse location.



More complex expressions (for example array fields) can be viewed by selecting them with the mouse and placing the mouse cursor over the selection.



The debugger tools also offer a number of ways to examine the content of variables , arrays or lists .

Errors in the Program

If the debugger encounters an error in your program, it will halt the execution, mark the line that contains the error (red background in the default colors) and display the error-message in the error log and the status bar.
At this point, you can still examine the variables of your program, the callstack or the memory, however other features like the Register display or stack trace are not available after an error.
If the error is determined to be fatal (like an invalid memory access, or division by 0), you are not allowed to continue the execution from this point. If the error was reported by a PureBasic library, you are allowed to try to continue, but in many cases, this may lead to further errors, as simply continuing just ignores the displayed error.
After an error (even fatal ones), you have to use the ”Kill Program” command to end the program and continue editing the source code. The reason why the program is not automatically ended is that this would not allow to use the other debugger features (like variable display) to find the cause of the error.
Note: you can configure the debugger to automatically kill the program on any error. See Customizing the IDE for that.

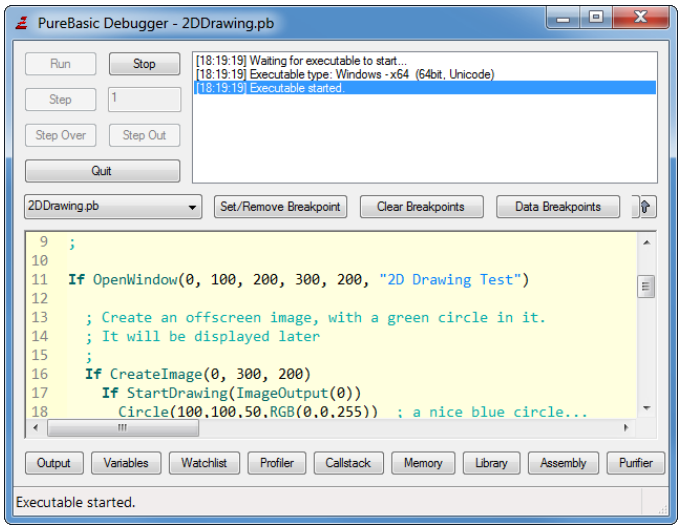
Debugger warnings

In some cases the debugger cannot be sure whether a given parameter is actually an error in the program or was specified like that on purpose. In such a case, the debugger issues a warning. By default, a warning will be displayed with file and line number in the error log and the line will be marked (orange in the default colors). This way the warnings do not go unnoticed, but they do not interrupt the program flow. There is also the option of either ignoring all warnings or treating all warnings like errors (stopping the program). The handling of debugger warnings can be customized globally in the Preferences or for the current compiled program in the Compiler options .

The Error log

The error log is used to keep track of the compiler errors, as well as the messages from the debugging. Messages are always logged for the file they concern, so when an error happens in an included file , this file will be displayed, and a message logged for it.
The ”Error log” submenu of the Debugger menu provides functions for that:
Show error log
Shows / hides the log for the current source.
Clear log
Clears the log for this file.
Copy log
Copies the contents of the error log to the clipboard.
Clear Error marks
After you have killed the program, any error mark in the source file will remain. This is to help you identify the line that caused the problem and solve it. The ”Clear error Marks” command can be used to remove these marks.
You can also configure the IDE to automatically clean the error marks when the program ends. See Configuring the IDE for that.

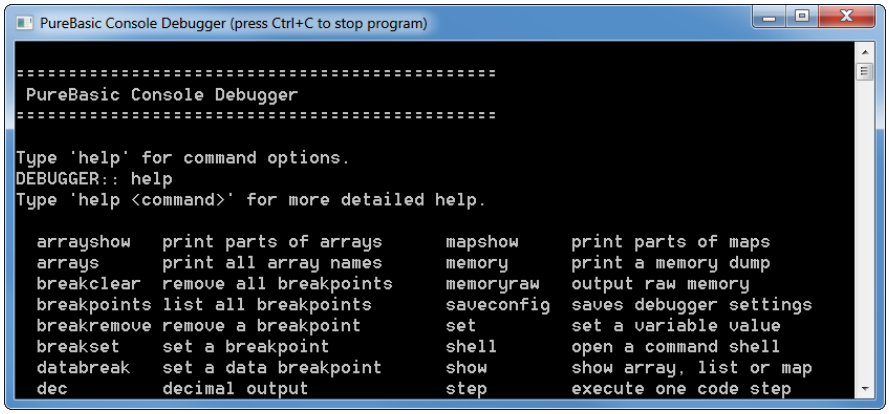
The Standalone Debugger



The standalone debugger is very similar to the one in the IDE, and will be explained here only briefly: On the Debugger window, you have control buttons to carry out the basic program control, as described above. The ”Step” button carries out as many steps as are set in the edit field next to it. Closing the Debugger with ”Quit” or the close button will also kill the debugged program.
The Error log area can be hidden by the up arrow button on the right side in order to make the debugger window smaller.
The code view is used to display the currently executed code line as well as any errors or breakpoints. Use the combo box above it to select the included file to view. The ”Set Breakpoint”, ”Remove

Breakpoint" and "Clear Breakpoints" can be used to manage breakpoints in the currently displayed source file. The code view also provides the mouse-over feature from the integrated debugger to quickly view the content of a variable.
The debugger tools can be accessed from the buttons below the code area. Their usage is the same as with the integrated IDE debugger.
Note: The Standalone Debugger has no configuration of its own. It will use the debugger settings and coloring options from the IDE. So if you use a third-party Editor and the standalone debugger, you should run the IDE at least once to customize the Debugger settings.
Executing the standalone debugger from the command-line:
To execute a program compiled on the command-line with enabled debugger (-d or /DEBUGGER switch), call the debugger like this:
pbdebugger <executable file> <executable command-line>
If you execute a debugger-enabled executable from the command-line directly, it will only use the command-line debugger.

The command-line debugger:



The command-line debugger is not a part of the IDE and therefore not explained in detail here.
While the program is running, hit Ctrl+C in the console to open a debugger console prompt. In this prompt type "help" to get an overview of all available commands. Type "help <commandname>" for a more detailed description of the command.

Debugging threaded programs:

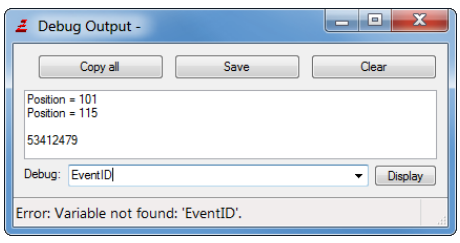
To use the debugger with a program that creates threads , the 'Create thread-safe executable' Option must be set in the Compiler options , as otherwise the information displayed by the debugger concerning line numbers, errors, local variables and such could be wrong due to the multiple threads.
The following features and limitations should be considered when debugging a threaded program:
While the program is running, the variable viewer, callstack display or assembly debugger will display information on the main thread only. When the program is stopped, they display information on the thread they were stopped in. So to examine local variables or the callstack of a thread, the execution must be halted within that thread (by putting a breakpoint or a CallDebugger statement there). The various 'Step' options always apply to the thread where the execution was last stopped in.
If an error occurs, the execution is halted within that thread, so any information displayed by the variable viewer or callstack display is of the thread that caused the error.
The watchlist only watches local variables of the main thread, not those of any additional running threads.
While the execution is stopped within one thread, the execution of all other threads is suspended as well.

Chapter 14

Included debugging tools

These tools provide many features to inspect your program while it is running. They can not be used while you are editing the source code. These tools are available in both the integrated Debugger and the standalone debugger. The console debugger provides many of these features too, but through a debugger console.
Some of the tools include the viewing of variables. Here is an explanation of the common fields of all these variable displays:
Scope
The scope of a variable is the area in which it is valid. It can be global , local , shared , static or threaded , depending on how it is used in your source code. 'byref' ("by reference", i.e. using the address) is used to indicate an Array or List that was passed as parameter to a procedure.
Variable type
The variable type is indicated through a colored icon:
B : Byte
A : Ascii
C : Character
W : Word
U : Unicode
L : Long
I : Integer
Q : Quad
F : Float
D : Double
S : String
Sn : Fixed length string
A Structure is either marked as a dot, or with an arrow. If marked with an arrow, it can be expanded by double-clicking on it to view the members of this structure. A down arrow marks an expanded structure. A Structure marked with a dot cannot be expanded (usually because it is just a structure pointer).
Dynamic arrays inside structures are shown with the dimensions they are currently allocated with. Lists and maps inside structures are shown with their size and their current element (if any).

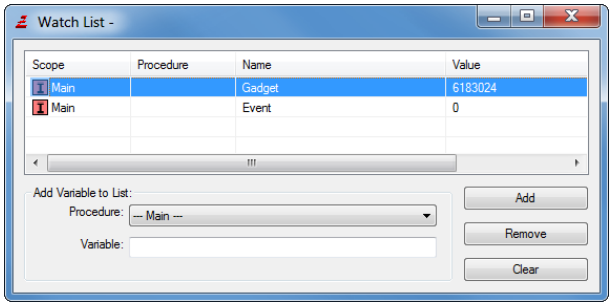
The Debug output window



In this window, the output of the Debug statement will be displayed. The Debug statement is a quick

and simply way to print messages for debugging purposes.
The debug window will automatically open at the first output is produced by your program. If you then close it, it will not automatically open on subsequent messages, however they will still be logged. You can copy this output to the clipboard or save it to a file. There is also a button to clear the messages from the window.
The entry field at the bottom of the window allows an expression to be entered, which will be evaluated and the result printed in the output box. This allows to quickly check the state of variables or array fields without the need to look them up in one of the debugger tools. The evaluation is started by pressing Enter or clicking on the "Display" button. If the expression cannot be evaluated for some reason, an error-message is displayed in the statusbar.
The expression can be any valid PB expression (not including logical ones or containing PB keywords). It can contain variables , arrays , lists , constants and also some commands from the Math , Memory and String libraries.

The Watchlist

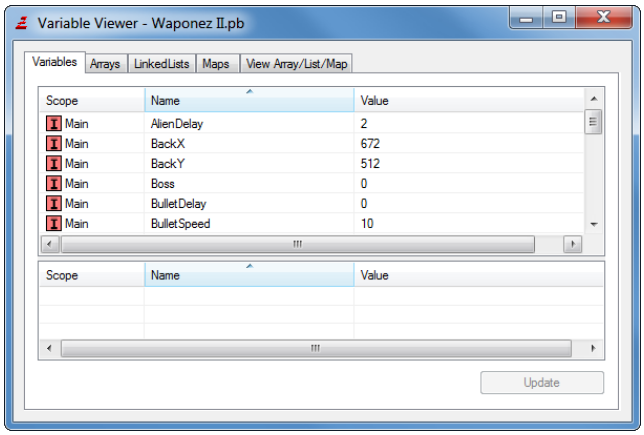


The watchlist can be used to track changes in variables of your program in real time, while the program is running. It can only display single variables (no full structures), however, these variables can be a part of structures. Elements of dynamic array, list or map inside structures can not be displayed in the watchlist.
To add a variable, select its procedure (if it is a local variable) or select "— Main —" if it is a global variable or part of an array or list . Then type the variable name, as you would access it in your source code into the Variable field and press Add.

```
Examples:
MyVariable$                    - add a normal string variable
Array(1, 5)                    - add an array field
Structure\subfield[5]\value    - add a variable inside a structure
MyList()\structuresubfield     - add a variable inside a structured list
```

You can also add new watched variables from the VariableViewer, by right-clicking on them and selecting "add to watchlist"
In the list you will see the values of the watched variables. If the value is displayed as "—", it means that this variable is not valid at the current point in the source code; for example, this will occur if you watch a local variable, or a list element and the list has no current element.
The watched variables are remembered between the debugging sessions, and even saved with the compiler options, so you do not need to repopulate this list all the time.

The Variable Viewer



The Variable viewer allows to examine the program's variables, arrays , lists and maps . The individual tabs show global and threaded items in the top part and local , shared and static items in the bottom part.
The "Update" Button can be used to get the most recent data from the program. If the program is halted or in step mode, the content is updated on each step automatically. By right-clicking on any variable or Array/List field, you can copy that variable, or add it to the watchlist for real-time tracking of its value. On Windows, the content of the Variable viewer can be sorted by name, scope or variable value by clicking on the header of the appropriate column.

The 'Variables' tab

This tab, shows the variables of the program. By right-clicking on a variable, it is possible to add it to the watchlist.

The 'Arrays' tab

This tab shows a list of all arrays in the program and the dimensions in which they are currently defined (-1 means that Dim was not called yet). By right-clicking on an array, the content of the array can be viewed in the "View Array/List/Map" tab.

The 'Lists' tab

This tab shows a list of all Lists, the number of elements they currently hold ("-" indicates that NewList was not called yet), as well as the index of the current element of the list ("-" indicates that there is no current element). By right-clicking on a list, the content of the list can be viewed in the "View Array/List/Map" tab.

The 'Maps' tab

This tab shows a list of all maps, the number of elements they currently hold ("-" indicates that NewMap was not called yet), as well as the key of the current element of the map ("-" indicates that there is no current element). By right-clicking on a map, the content of the map can be viewed in the "View Array/List/Map" tab.

The 'View Array/List/Map' tab

This tab can be used to view individual entries of an array, a list or a map. This includes arrays, lists or maps inside structures as well. To do so enter the name of the array, map or list including a "()" at the end, select what kind of items to display and press "Display". Note that the content of the display is not automatically updated when the program is in step mode.
"Display all items" simply displays everything. "Display Non-zero items only" will only display those items that do not hold the value 0 or an empty string. This makes viewing large arrays/lists with only few valid items in them simpler. A structure is considered "zero" if all of its items either hold the value 0 or an empty string.
"Display Range" allows displaying only a specific range of an array, a list or a map. The range can be given for each Array dimension individually, separated by commas. If one dimension is not specified at all, all of its items will be displayed. Here are a few examples for valid range input:

```
"1-2, 2-5, 2" : first index between 1 and 2, a second index between 2
 and 5 and a third index of 2.
"1, 2-5"      : first index of 1 and a second index between 2 and 5.
"1, , 5"      : first index of 1, any second index and a third index
 of 5.
```

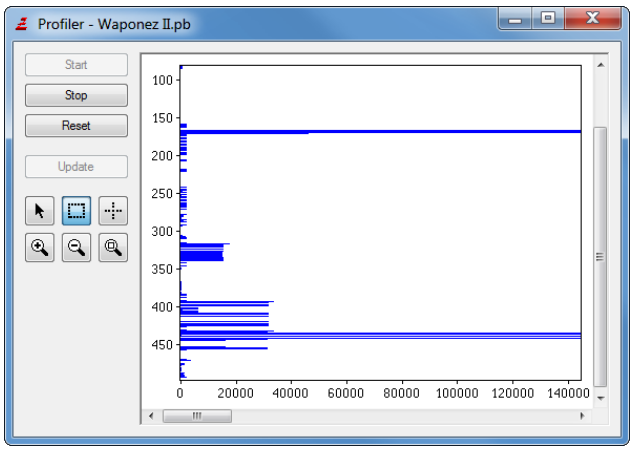
For list display, the ”Display Range” option can be used to display a range of list elements by their index (zero based).

```
"0"     : first element
"1-3"   : second to the fourth element
```

For map display, the ”Display Range” option can be used to filter the keys to be displayed. It has to contain a mask for the key string of the map elements (no quotation marks). A ”?” matches one character, a ”*” matches any number of characters. Here are a few examples for valid mask input:

```
"hat" : matches only the item with "hat" as key.
"?at" : matches items with "hat", "bat" etc as key.
"h*t" : matches items with keys that start with "h" and end with "t"
 and anything in between.
```

The Profiler



The profiler tool can count how often each line in the source code is executed. This collected data can be used to identify which portions of the code are used most often and where improvements make most sense. It also helps to identify problems where a piece of the code is executed too often as a result of an error.

Recording the data

The recording of the data can be controlled by the Start, Stop and Reset (to set all counts to 0) buttons in the profiler window. The Update button can be used to update the graph while the program is running. Each time the program is halted or in step mode, the data is updated automatically. By default, the profiler is recording data from the start of the program. This can be changed in the Preferences .

Examining the data

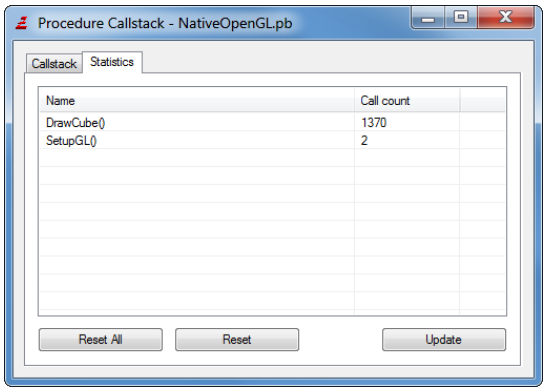
The recorded data is displayed as a graph, with the vertical axis showing the source line number and the horizontal axis showing how often the lines were executed. If the running program consists of more than one source file, a list of source files is presented below the graph. To display the graph for a file either select it, or check its checkbox. Multiple files can be shown in the graph at once to better compare them. Right-clicking on one of the filenames allows changing the color used to display that file in the graph.

Mouse modes in the graph

Right-clicking inside the graph brings up a popupmenu which allows zooming in or out or to show the source line that was clicked on in the IDE or code display of the debugger. The action taken by a left-click can be controlled by the buttons on the left side:

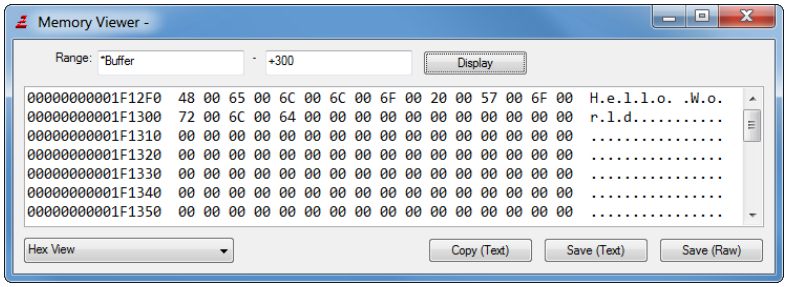
- Arrow button: Left-clicking and dragging allows to scroll the graph display.
- Box button: Left-clicking and dragging allows to select an area which will be zoomed in.
- Cross button: While this button is activated, moving the mouse on the graph displays a crosshair to better identify the line and call count under the mouse.
- Zoom buttons: These buttons allow to zoom in/out and zoom out so all lines can be viewed.

The Callstack viewer



The callstack viewer shows which nested procedure calls led to the current position in the code. Each entry in the list means one procedure that is currently open. It shows the line and file from which it was called, and the arguments used to call the procedure. By clicking on the Variables button for each procedure, you can look at the variables of that instance of the procedure.
This allows to easily trace, from which part of the code, a procedure was called. The callstack view does only automatically update when you stop the program, or use Step to execute single lines. While the program is running, you have to use the Update button to update the view for the current code position.
The ”Statistics” tab shows the number of times each procedure in the code was called. You can reset the count for all procedures with ”Reset all”, or for the currently marked entry with the ”Reset” button. Like with the callstack, the updates are not automatic while the program is not stopped. Use the Update button for that.

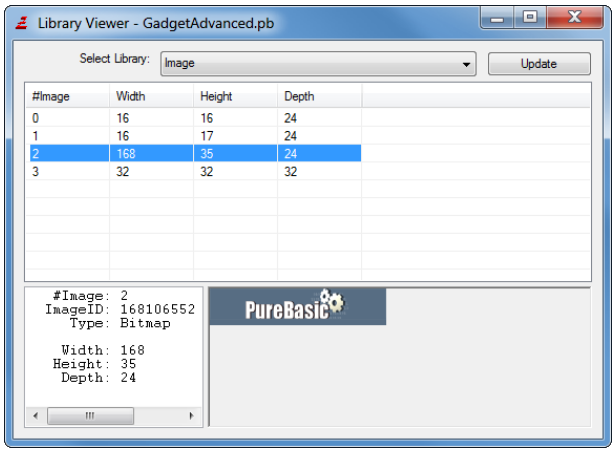
The Memory Viewer



The memory viewer lets you view a memory area in your program. The range to view can be entered into the range fields as any valid PureBasic expression; this can be a normal decimal value, a hex number preceded by the $ character or any other valid expression, including variables or pointers from the code. If the content of the second range field starts with a ”+” sign, the content is interpreted as relative to the first field.
Example: ”*Buffer + 10” to ”+30” will display the 30 bytes of memory starting at the location 10 bytes after what *Buffer points to.
If the memory area is valid for viewing, it will be displayed in the area below. If parts of the area are not valid for reading, you will get an error-message. The type of display can be changed with the combo box in the lower left corner. The following view modes are available:
Hex View
The memory will be displayed like in any hex viewer, giving the memory location in hex display on the left, followed by the hexadecimal byte values, and then the string representation in the right column.
Byte/Character/Word/Long/Quad/Float/Double table
The memory area will be shown as a table of the specified variable type. Whether or not this table is single-column or multi-column can be set in the Preferences (see Configuring the IDE). Data will be shown as decimal, octal or hex via the next button that cycles the mode.
String view ascii, unicode or utf-8
This displays the memory area as a string, with any non-string characters displayed in [] (for example ”[NULL]” for the 0 byte.) A linebreak is added after newline characters and [NULL] to improve the

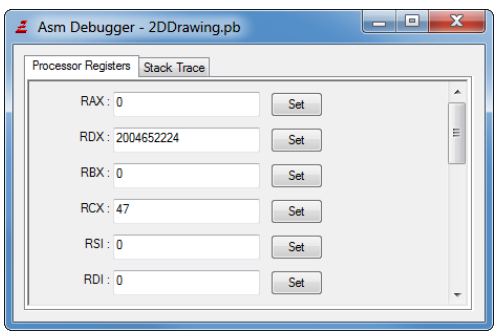
readability of the output. The memory area can be interpreted as an Ascii, Unicode or Utf8 string.
Data
With the Data check box, you will export the table view formatted as a Data Section on copy or save.
You can also export the viewed memory area from the memory viewer:
Copy (Text): Copies the displayed area as text to the clipboard.
Save (Text): Saves the displayed area as text to a file.
Save (Raw): Saves the memory area as raw binary to a file.

The Library Viewer



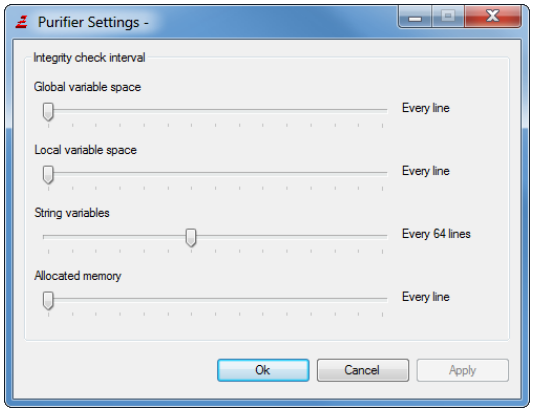
The Library Viewer provides information about the objects created by some libraries. For example, it allows to quickly check which images are currently loaded in the program, or which gadgets are created. Once the program is started, the combobox on top of the window can be used to select the library to view. The list below will then show all objects of the library that currently exist in the executable along with some additional information on each object. The "Update" button will update this list of objects. Selecting an object in the list will display more detailed information about it in the text area on the left, and if supported by the library also a visual display of the object on the right (for Images, Sprites, and so on).
If the combobox displays "No Information", this means that your executable has used no library that supports this feature.
Currently, the Library Viewer is supported by the following libraries:
Thread
Gadget
Window
File
Image
Sprite
XML

The Assembly Debugger



The ASM debugger is provided for advanced programmers to examine and change the CPU register content and to examine the programs stack for debugging of inline ASM code.
The Processor Register view is only available while the program execution is halted. By changing any of the register values and clicking ”Set”, you can modify the value in that register.
The Stack trace shows the content of the programs stack given in relation to the ESP register. If the current stack position is not aligned at a 4 byte boundary, there can be no information given about the content of the stack. In this case the stack is shown as a hex display.
If the stack pointer is properly aligned, the stack contents are displayed with comments about the meaning of the contained values (detailing the pushed registers and other values on a PureBasic procedure call).
The stack trace is updated automatically when you stop the execution or step though the program, unless you specify otherwise in the Preferences . If you disable the automatic update, there will be an ”Update” button displayed to do so manually.
Note: The Assembly debugger is currently not available on MacOS X.

The Purifier



The purifier can detect memory errors such as writing past the end of an allocated memory buffer or string. Without the purifier some of these mistakes would lead to crashes and others would go unnoticed because the write operation overwrites some other valid memory.
The purifier requires special output from the compiler to work, which is why it is only available if the ”Enable Purifier” option is set in the compiler options when the program is compiled.
The purifier detects these errors by placing a special ’salt’-value around global and local variables, strings and allocated memory buffers. These salt values are then checked at regular intervals and an error is displayed if they were changed. These checks slow down the program execution considerably especially for large programs, which is why the rate at which the checks are performed can be specified in the purifier window:
Global variable space
Defines the interval in source lines after which the global variables are checked.
Local variable space
Defines the interval in source lines after which the local variables are checked.
String variables
Defines the interval in source lines after which the memory used by string variables is checked.
Allocated memory
Defines the interval in source lines after which the memory allocated by AllocateMemory() is checked.

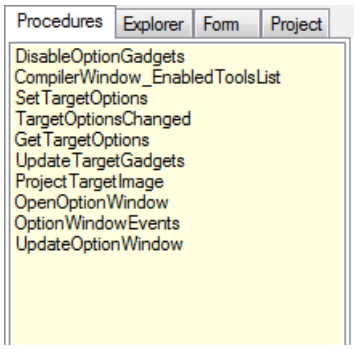
Chapter 15

Using the built-in Tools

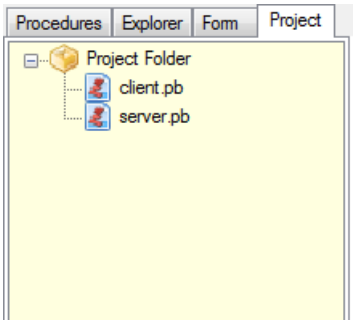
The PureBasic IDE comes with many building tools, to make programming tasks easier and increase your productivity. Many of them can be configured to be either accessible from the Menu as separate windows, or to be permanently displayed in the Panel on the side of the editing area.
For information on how to configure these tools and where they are displayed, see Configuring the IDE .

Tools for the Side Panel Area

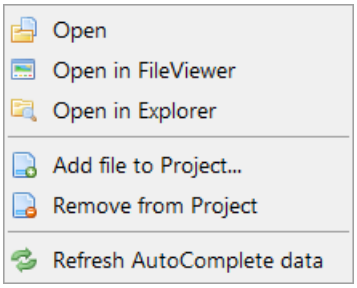
Procedure Browser



This tool displays a list of all procedures and macros declared in the current source code. By double-clicking on an entry in that list, the cursor automatically jumps to that procedure.
Macros will be marked in the list by a "+" sign before the name.
You can also place special comment marks in your code, that will be displayed in the list too. They look like this: ";- <description>". The ; starts a comment, the - that follows it immediately defines such a mark.
The description will be shown in the Procedure list, and clicking on it will jump to the line of this mark. Such a comment mark can be distinguished from a Procedure by the "> " that is displayed before it in the procedure list.
The list of procedures can be sorted, and it can display the procedure/macro arguments in the list. For these options, see Configuring the IDE .
Project Panel

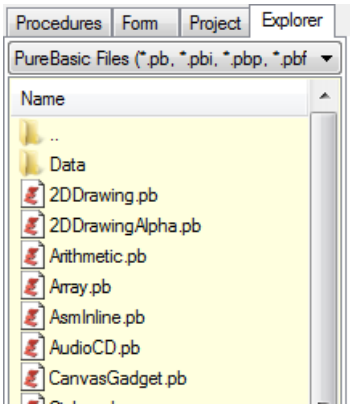


This tool displays a tree of all files in the current project . A double-click on a file opens it in the IDE. This allows fast access to all files in the project. A right-click on a file opens a context menu which provides more options:



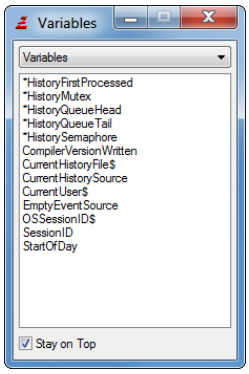
Open - Open the file in the IDE.
Open in FileViewer - Open the file in the FileViewer of the IDE.
Open in Explorer - Open the file in the operating systems file manager.
Add File to Project - Add a new file to the project.
Remove File from Project - Remove the selected file(s) from the project.
Refresh AutoComplete data - Rescan the file for AutoComplete items.

Explorer



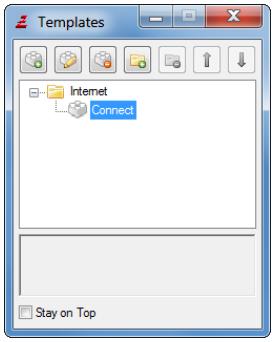
The Explorer tool displays an explorer, from which you can select files and open them quickly with a double-click. PureBasic files (*.pb, *.pbi, *.pbp, *.pbf) will be loaded into the edit area and all other recognized files (text & binary) files will be displayed into the internal File Viewer.

Variable Viewer



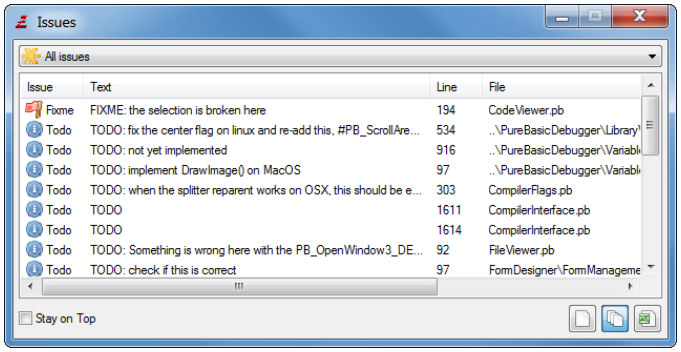
The variable viewer can display variables , Arrays , lists , Constants , Structures and Interfaces defined in your source code, or any currently opened file. You can configure what exactly it should display in the preferences .
Note: The displaying of variables is somewhat limited for now. It can only detect variables explicitly declared with Define , Global , Shared , Protected or Static .

Code Templates



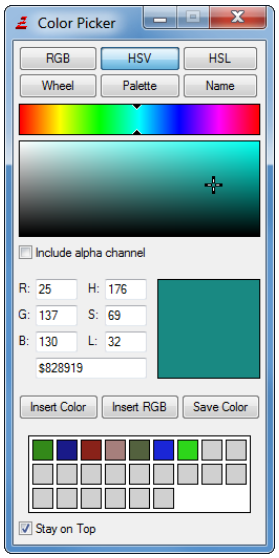
The templates tool allows you to manage a list of small code parts, that you can quickly insert into your source code with a double-click. It allows you to manage the codes in different directories, and put a comment to each code. This tool is perfect to manage small, often used code parts.

Issue Browser



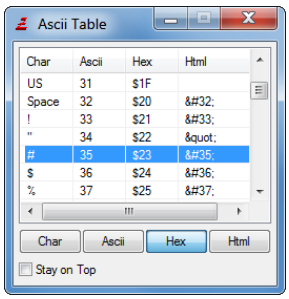
The issue browser tool collects comments in the source code that fit a defined format and lists them ordered by priority. It can be used to track which areas of the source code still need to be worked on. Each displayed issue corresponds to one comment in the code. A double-click on the issue shows that code line. Issues can be displayed for the current file, or for multiple files (all open files, or all files that belong to the current project). The issue list can also be exported in CSV format.
To configure the collected issues, see the ”Issues” section in the Preferences .

Color Picker



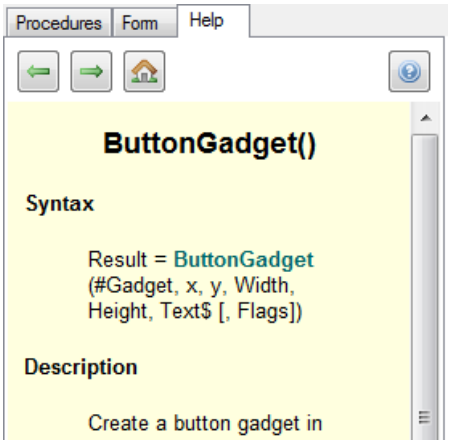
The color picker helps you to find the perfect color value for whatever task you need. The following methods of picking a color are available:
RGB: Select a color by choosing red, green and blue intensities.
HSV: Select a color by choosing hue, saturation and value.
HSL: Select a color by choosing hue, saturation and lightness.
Wheel: Select a color using the HSV model in a color wheel.
Palette: Select a color from a predefined palette.
Name: Select a color from a palette by name.
The color selection includes an alpha component, if the ”Include alpha channel” checkbox is activated. The individual components (red/green/blue intensities or hue/saturation/lightness) as well as the hexadecimal representation of the current color can be seen and modified in the text fields.
The ”Insert Color” button inserts the hexadecimal value of the current color in the source code. The ”Insert RGB” button inserts the color as a call to the RGB() or RGBA() function into the code. The ”Save Color” button saves the current color to the history area at the bottom. Clicking on a color in the history makes it the current color again.

Character Table



The character table tool displays a table showing the first 256 unicode characters, together with their index in decimal and hex, as well as the corresponding html notation. By double-clicking on any line, this character will be inserted into the source code. With the buttons on the bottom, you can select which column of the table to insert on a double-click.

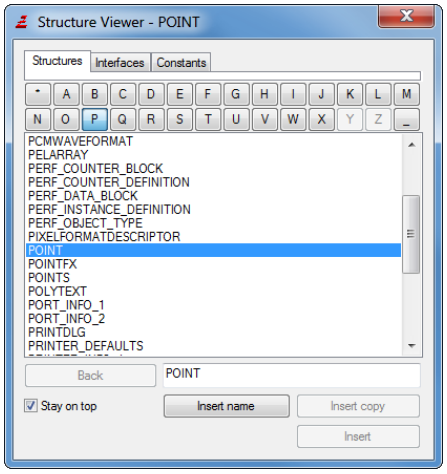
Help Tool



The Help Tool is an alternative viewer for the reference guide . It can be used to view the PureBasic manual side by side with the code. Whether or not the F1 shortcut opens the manual in the tool or as a separate window can be specified in the preferences .

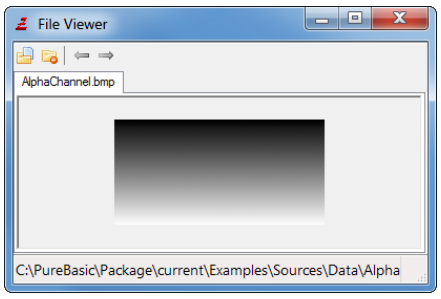
Other built-in tools

Structure Viewer



The structure viewer allows you to view all the Structures, Interfaces and Constants predefined in PureBasic. Double-clicking on a Structure or Interface shows the declaration. On top of the list you can select a filter to display only entries that start with a given character.
The ”Back” button navigates back through the viewed entries.
”Insert name” inserts just the name of the selected entry.
”Insert copy” inserts a copy of the declaration of that entry.
”Insert” lets you enter a variable name and then inserts a definition of that variable and the selected entry and all elements of it.

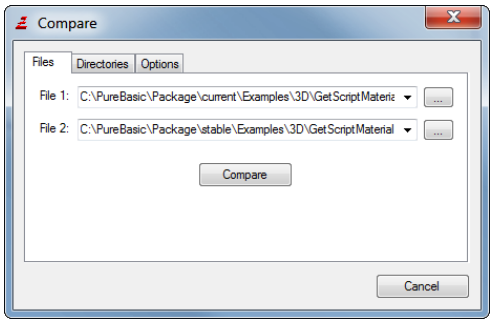
File Viewer



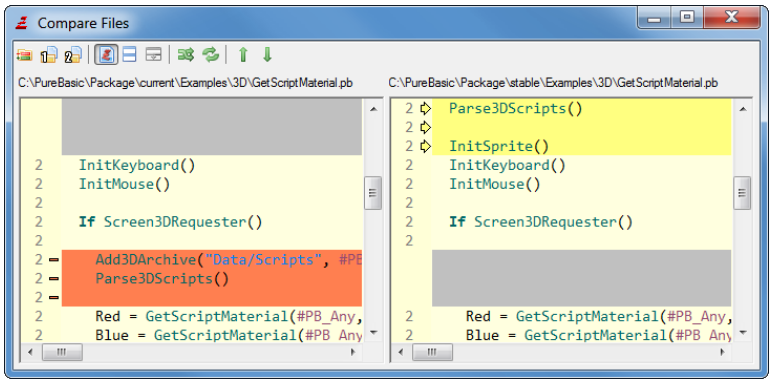
The internal file viewer allows you do display certain types of files. Text files, images and web pages

(windows only). Any unknown file type will be displayed in a hex-viewer. The ”Open” button opens a new file, the ”X button” closes it and the arrows can be used to navigate through the open files.
Also any binary file that you attempt to open from the Explorer tool, or by double-clicking on an IncludeBinary keyword will be displayed in this file viewer.

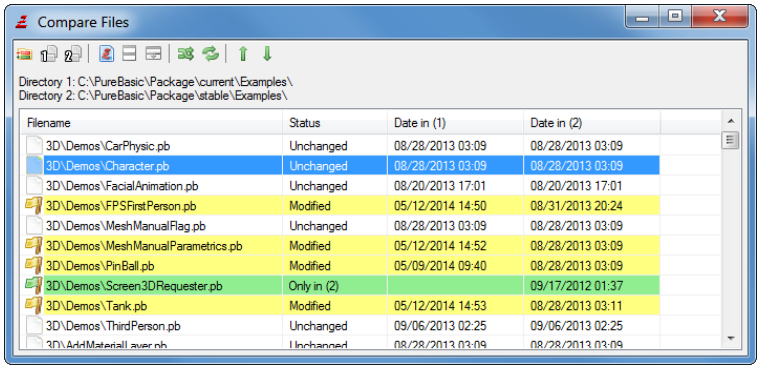
Compare Files/Folders



This tool can compare two (text-) files or two directories and highlight their differences. The ”Options” tab can be used to ignore some differences such as spaces or upper/lowercase changes.



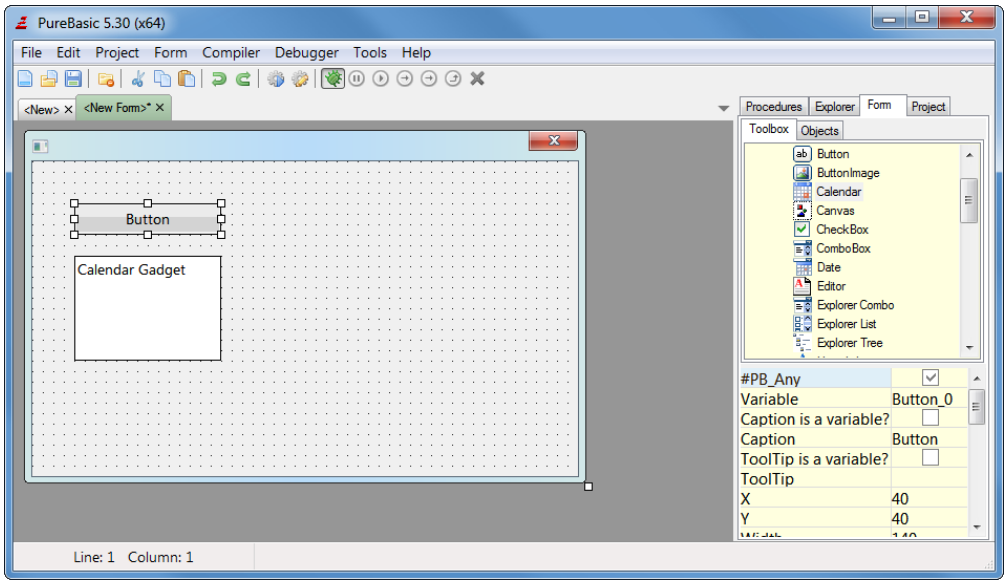
The files are shown side by side with the differences marked in the following way: Lines shown in red were removed in the file on the right, lines shown in green were added in the file on the right and lines shown in yellow were changed between the two files.



When comparing directories, the content of both directories is examined (with the option to filter the search by file extension and include subdirectories) and the files are marked in a similar way: Files in red do not exist in the second directory, files in green are new in the second directory and files in yellow were modified. A double-click on a modified file shows the modifications made to that file.

Other entries in the Tools menu

Form Designer

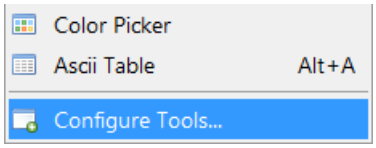


The Form Designer can be used to design the user interface for you application. For more information, see the form designer chapter.

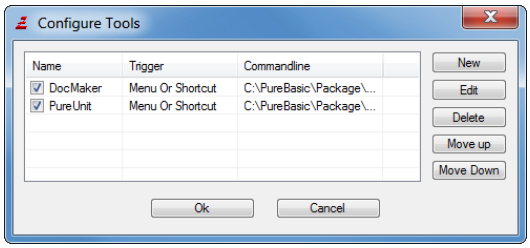
Chapter 16

Using external tools

The PureBasic IDE allows you to configure external programs to be called directly from the IDE, through the Menu, Shortcuts, the Toolbar, or on special "triggers". The use of this is to make any other program you use while programming easily accessible.
You can also write your own little tools in PureBasic that will perform special actions on the source code you are currently viewing to automate common tasks. Furthermore, you can configure external file viewers to replace the internal File Viewer of the IDE for either specific file types or all files.



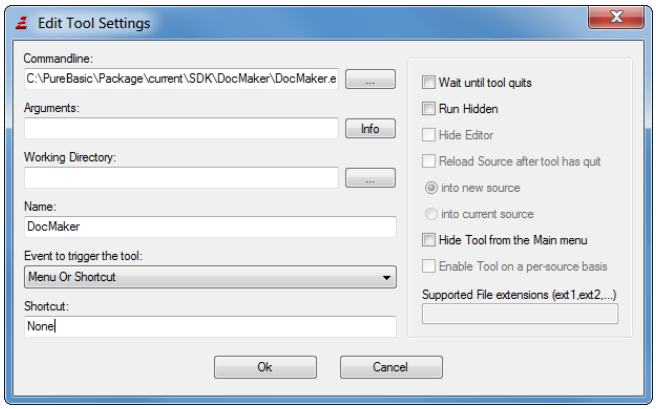
With the "Config tools" command in the Tools menu, you can configure such external tools. The list you will see displays all the configured tools in the order they appear in the Tools menu (if not hidden). You can add and remove tools here, or change the order by clicking "Move Up"/"Move Down " after selecting an item.



Any tool can be quickly enabled or disabled from the "Config tools" window with the checkbox before each tool entry. A checked checkbox means the tool is enabled, an unchecked one means it is currently disabled.

Configuring a tool

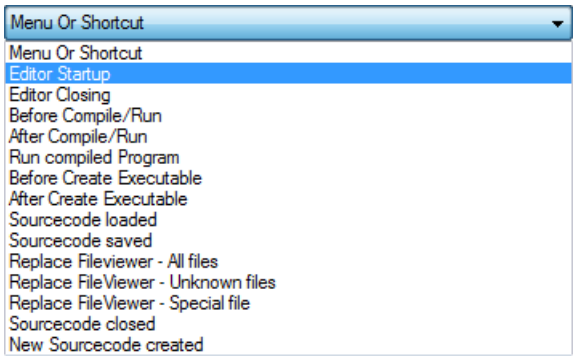
The basic things you need to set is the command-line of the program to run, and a name for it in the Tools list/Menu. Everything else is optional.



Command-line
Select the program name to execute here.
Arguments
Place command-line arguments that will be passed to the program here. You can place fixed options, as well as special tokens that will be replaced when running the program:
%PATH : will be replaced with the path of the current source code. Remains empty if the source was not saved.
%FILE : filename of the current source code. Remains empty if it has not yet been saved. If you configure the tool to replace the file viewer, this token represents the file that is to be opened. You should never modify the contents of the %FILE file. This file contains the state of the source code as the user last saved it. This could be much different from what is currently visible in the editor. If you overwrite this file, the user might lose data.
%TEMPFILE : When this option is given, the current source code is saved in a temporary file, and the filename is inserted here. You may modify or delete the file at will.
%COMPILEFILE : This token is only valid for the compilation triggers (see below). This is replaced with the temporary file that is sent to the compiler for compilation. By modifying this file, you can actually change what will be compiled.
%EXECUTABLE : This will be replaced by the name of the executable that was created in with the last "Create Executable". For the "After Compile/Run" trigger, this will be replaces with the name of the temporary executable file created by the compiler.
%CURSOR : this will be replaced by the current cursor position in the form of LINExCOLUMN.
%SELECTION : this will be replaced by the current selection in the form of LINESTARTxCOLUMNSTARTxLINEENDxCOLUMNEND. This can be used together with %TEMPFILE, if you want your tool to do some action based on the selected area of text.
%WORD : contains the word currently under the cursor.
%PROJECT : the full path to the directory containing the project file if a project is open.
%HOME : the full path to the purebasic directory
Note: for any filename or path tokens, it is generally a good idea to place them in "" (i.e. "%TEMPFILE") to ensure also paths with spaces in them are passed correctly to the tool. These tokens and a description can also be viewed by clicking the "Info" button next to the Arguments field.
Working Directory
Select a directory in which to execute this tool. By specifying no directory here, the tool will be executed in the directory of the currently open source code.
Name
Select a name for the tool. This name will be displayed in the tools list, and if the tool is not hidden from the menu, also in the Tools menu.

Event to trigger the tool

Here you can select when the tool should be executed. Any number of tools can have the same trigger, they will all be executed when the trigger event happens. The order of their execution depends on the order they appear in the tools list.



Menu Or Shortcut
The tool will not be executed automatically. It will be run by a shortcut or from the Menu. Note: to execute a tool from the Toolbar, you have to add a button for it in the Toolbar configuration in the Preferences (see Configuring the IDE for more).
With this trigger set, the "Shortcut" option below becomes valid and lets you specify a shortcut that will execute this tool.
Editor Startup
The tool will be executed right after the IDE has been fully started.
Editor End
The tool will be executed right before the IDE ends. Note that all open sources have already been closed at this time.
Before Compile/Run
The tool will be executed right before the compiler is called to compile a source code. Using the %COMPILEFILE token, you can get the code to be compiled and modify it. This makes it possible to write a small pre-processor for the source code. Note that you should enable the "Wait until tool quits" option if you want your modifications to be given to the compiler.
After Compile/Run
The tool will be executed right after the compilation is finished, but before the executable is executed for testing. Using the %EXECUTABLE token, you can get access to the file that has just been created. Note that you can modify the file, but not delete it, as that results in an error-message when the IDE tries to execute the file.
Run compiled Program
The tool will be executed when the user selects the "Run" command from the compiler menu. The tool is executed before the executable is started. The %EXECUTABLE token is valid here too.
Before create Executable
The same as for the "Before Compile/Run" trigger applies here too, only that the triggering event is when the user creates the final executable.
After create Executable
The tool is executed after the compilation to create the final executable is complete. You can use the %EXECUTABLE token to get the name of the created file and perform any further action on it.
Source code loaded
The tool is executed after a source code has been loaded into the IDE. The %FILE and %PATH tokens are always valid here, as the file was just loaded from the disk.
Source code saved
The tool will be executed after a source code in the IDE has been saved successfully. The %FILE and %PATH tokens are always valid here, as the file has just been saved to disk.
Source code closed
The tool will be executed whenever a source file is about to be closed. At this point the file is still there, so you can still get its content with the %TEMPFILE token. %FILE will be empty if the file was never saved.
File Viewer All Files
The tool will completely replace the internal file viewer. If an attempt is made in the IDE to open a file that cannot be loaded into the edit area, the IDE will first try the tools that have a trigger set for the specific file type, and if none is found, the file will be directed to this tool. Use the %FILE token to get the filename of the file to be opened.
Note: Only one tool can have this trigger. Any other tools with this trigger will be ignored.
File Viewer Unknown file

This tool basically replaces the hex viewer, which is usually used to display unknown file types. It will be executed, when the file extension is unknown to the IDE, and if no other external tool is configured to handle the file (if a tool is set with the "File Viewer All Files" trigger, then this tool will never be called). Note: Only one tool can have this trigger set.

File Viewer Special file

This configures the tool to handle specific file extensions. It has a higher priority than the "File Viewer All files" or "File Viewer Unknown file" triggers and also higher than the internal file viewer itself. Specify the extensions that the tool should handle in the edit box on the right. Multiple extensions can be given.

A common use for this trigger is for example to configure a program like Acrobat Reader to handle the "pdf" extension, which enables you to easily open pdf files from the Explorer, the File Viewer, or by double-clicking on an Includebinary statement in the source.

Open File with specific extension

This will be triggered for specific file extensions. It has a higher priority than the "Open File non-PB binary file" or "Open File non-PB text file" triggers. Specify the extensions that the tool should handle in the edit box on the right. Multiple extensions can be given.

Note: The Open File triggers will be active when you open a file via the File/Open menu and also, when you drag and drop a file to the IDE. They have higher priority for File/Open tasks. Only if there is no active tool, the File Viewer triggers will be handled.

Open File non-PB binary file

This one will be triggered for binary files, which are not part of PureBasic (more or less any)

Note: Only one tool can have this trigger. Any other tools with this trigger will be ignored.

Open File non-PB text file

This one will be triggered for text files, which are not part of PureBasic

Note: Only one tool can have this trigger set.

Other options on the right side

Wait until tool quits

The IDE will be locked for no input and cease all its actions until you tool has finished running. This option is required if you want to modify a source code and reload it afterwards, or have it passed on to the compiler for the compilation triggers.

Run hidden

Runs the program in invisible mode. Do not use this option for any program that might expect user input, as there will be no way to close it in that case.

Hide editor

This is only possible with the "wait until tool quits" option set. Hides the editor while the tool is running.

Reload Source after the tool has quit

This is only possible with the "wait until tool quits" option set, and when either the %FILE or %TEMPFILE tokens are used in the Arguments list.

After your program has quit, the IDE will reload the source code back into the editor. You can select whether it should replace the old code or be opened in a new code view.

Hide Tool from the Main menu

Hides the tool from the Tools menu. This is useful for tools that should only be executed by a special trigger, but not from the menu.

Enable Tool on a per-source basis

Tools with this option set will be listed in the "Execute tools" list in the compiler options , and only executed for sources where it is enabled there. Note that when disabling the tool with the checkbox here in the "Config tools" window, it will be globally disabled and not run for any source code, even if enabled there.

This option is only available for the following triggers:

- Before Compile/Run
- After Compile/Run
- Run compiled Program
- Before create Executable
- After create Executable
- Source code loaded
- Source code saved

- Source code closed

Supported File extensions

Only for the "File Viewer Special file" trigger. Enter the list of handled extensions here.

Tips for writing your own code processing tools

The IDE provides additional information for the tools in the form of environment variables. They can be easily read inside the tool with the commands of the Process library .

This is a list of provided variables. Note that those that provide information about the active source are not present for tools executed on IDE startup or end.

```
PB_TOOL_IDE          - Full path and filename of the IDE
PB_TOOL_Compiler     - Full path and filename of the Compiler
PB_TOOL_Preferences  - Full path and filename of the IDE's Preference
 file
PB_TOOL_Project      - Full path and filename of the currently open
 project (if any)
PB_TOOL_Language     - Language currently used in the IDE
PB_TOOL_FileList     - A list of all open files in the IDE, separated
 by Chr(10)

PB_TOOL_Debugger     - These variables provide the settings from the
 Compiler Options

PB_TOOL_InlineASM      window for the current source. They are set to
 "1" if the option
PB_TOOL_Unicode        is enabled, and "0" if not.
PB_TOOL_Thread
PB_TOOL_XPSkin
PB_TOOL_OnError

PB_TOOL_SubSystem    - content of the "Subsystem" field in the
 compiler options
PB_TOOL_Executable   - same as the %COMPILEFILE token for the
 command-line
PB_TOOL_Cursor       - same as the %CURSOR token for the command-line
PB_TOOL_Selection    - same as the %SELECTION token for the
 command-line
PB_TOOL_Word         - same as the %WORD token for the command-line

PB_TOOL_MainWindow   - OS handle to the main IDE window
PB_TOOL_Scintilla    - OS handle to the Scintilla editing component of
 the current source
```

When the %TEMPFILE or %COMPILEFILE tokens are used, the IDE appends the compiler options as a comment to the end of the created temporary file, even if the user did choose to not save the options there when saving a source code.

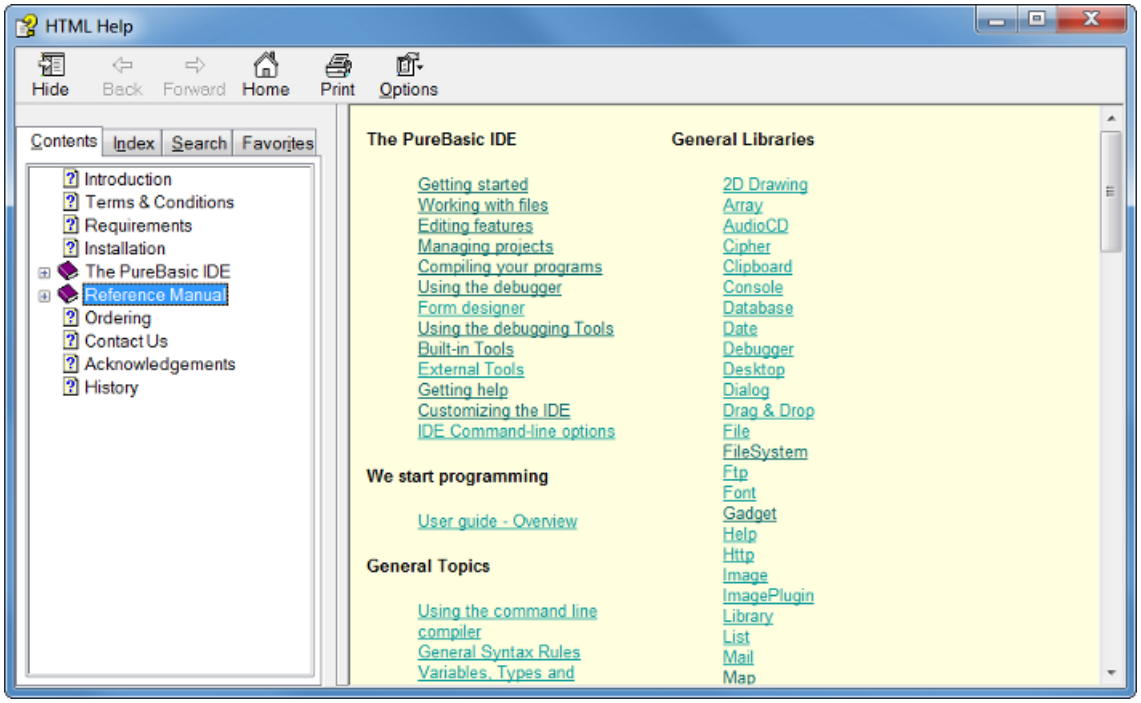
This enables your tool to read the compiler settings for this file, and take them into account for the actions your carries out.

Chapter 17

Getting Help

The PureBasic IDE provides ways to access the PureBasic help-file, as well as other files and documentation you want to view while programming.

Quick access to the reference guide



By pressing the help shortcut (F1 by default) or selecting the ”Help...” command from the Help menu while the mouse cursor is over a PureBasic keyword or function, the help will be opened directly at the description of that keyword or function.
If the word at the cursor position has no help entry, the main reference page will be displayed.
The reference manual can also be viewed side by side with the source code using the Help Tool .

Quick access to Windows API help

There are two ways to get the same quick access as for the PureBasic functions (by pressing F1 with the cursor on the function) also for functions of the Windows API. To enable this feature you have to download additional help files for these functions:

Microsoft Platform SDK The Microsoft Platform SDK provides the most complete and up to date programming reference available for the windows platform. It provides information on all API functions, as well as overviews and introductions to the different technologies used when programming for the windows platform. It is however quite big (up to 400MB depending on the selected components).
For the IDE help, you can either install the "February 2003" or the "Windows Server 2003 SP1" edition of the SDK.
It can be downloaded from here:
https://www.microsoft.com/en-us/download/details.aspx?id=15656
Note that the SDK can also be ordered on CD from this page. Also if you are the owner of any development products from Microsoft (such as Visual Studio), there might be a copy of the SDK included on one of the CDs that came with it.
The win32.hlp help-file There is a much smaller alternative to the complete SDK by Microsoft (7.5 MB download). This help is quite old (written for Windows95 in fact), so it does not provide any information on new APIs and technologies introduced since then.
However, it provides good information about commonly used API that is still valid today, as these mostly did not change. This download is recommended if you only need occasional help for API functions, but do not want to download the full SDK.
It can be downloaded from here:
http://www.purebasic.com/download/WindowsHelp.zip
To use it from the PureBasic IDE, just create a "Help" subdirectory in your PureBasic folder and copy the "win32.hlp" file into it.

Accessing external helpfiles from the IDE

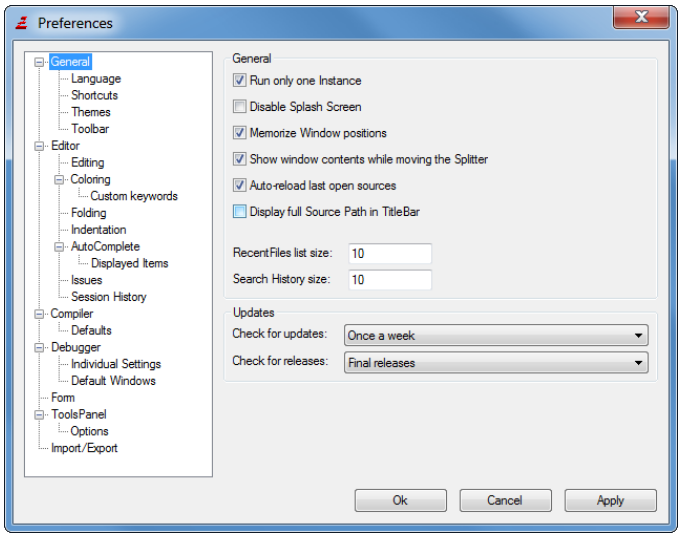
If you have other helpfiles you wish to be able to access from the IDE, then create a "Help" subdirectory in your PureBasic folder and copy them to it. These files will appear in the "External Help" submenu of the Help menu, and in the popupmenu you get when right-clicking in the editing area. Chm and Hlp files will be displayed in the MS help viewer. The IDE will open the helpfiles in the internal fileviewer. So files like text files can be viewed directly like this. For other types, you can use the Config Tools menu to configure an external tool to handle the type of help-file you use. The help will then be displayed in that tool.
For example, if you have pdf helpfiles, configure an external tool to handle pdf files and put the files in the Help subdirectory of PureBasic. Now if you click the file in the "external help" menu, it will be opened in that external tool.

Chapter 18

Customizing the IDE

The PureBasic IDE provides many options to customize or disable some of its features in order to become the perfect tool for you.
These options are accessible from the Preferences command in the File menu, and the meaning of each setting is described here.
Any changes made will only take effect once you click the ”OK” button or ”Apply”.

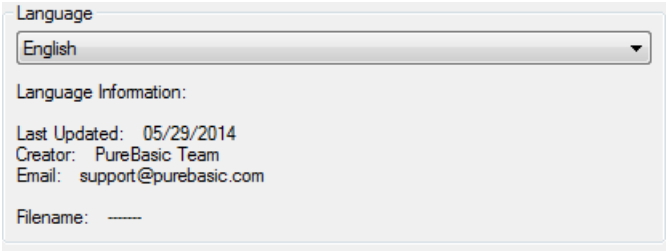
General



Options that affect the general behavior of the IDE.
Run only one Instance
If set, prevents the IDE from being opened more than once. Clicking on a PB file in the explorer will open it in the already existing IDE instance instead of opening a new one.
Disable Splash screen
Disables the splash screen that is displayed on start-up.
Memorize Window positions
Remembers the position of all IDE windows when you close them. If you prefer to have all windows open at a specific location and size, enable this option, move all windows to the perfect position, then restart the IDE (to save all options) and then disable this option to always open all windows in the last saved position.
Show window contents while moving the Splitter
Enable this only if you have a fast computer. Otherwise moving the Splitter bar to the Error Log or Tools Panel may flicker a lot.
Auto-Reload last open sources

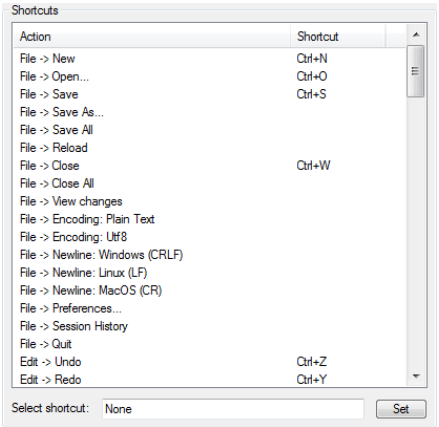
On IDE start-up, opens all the sources that were open when the IDE was closed the last time.
Display full Source Path in Title bar
If set, the IDE title bar will show the full path to the currently edited file. If not, only the filename is shown.
Enable Appearance Dark Mode
Is set, allows the appearance 'Dark Mode'.
Recent Files list
This setting specifies how many entries are shown in the "Recent Files" submenu of the File menu.
Search History size
This setting specifies how many recent search words are remembered for "Find/Replace" and "Find in Files"
Check for updates
Specifies how often the IDE should check on the purebasic.com server for the availability of new updates. An update check can also be performed manually at any time from the "Help" menu.
Check for releases
Specifies which kind of releases should cause a notification if they are available.

General - Language



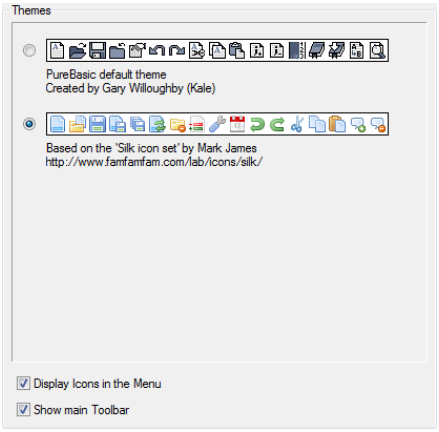
This allows you to change the language of the IDE. The combo box shows the available languages, and you can view some information about the language file (for example who translated it and when it was last updated).

General - Shortcuts



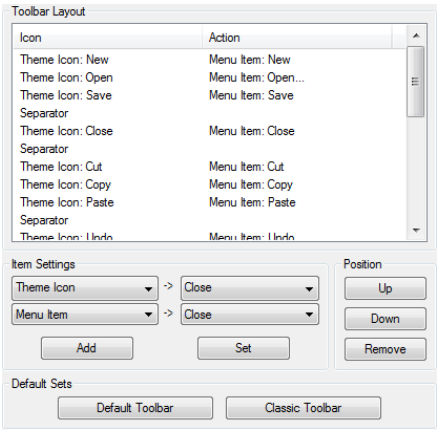
Here you can fully customize all the shortcut commands of the IDE. Select an entry from the list, select the shortcut field, enter the new key combination and click "Set" to change the entry.
Note that Tab & Shift+Tab are reserved for block-indentation and un-indentation and cannot be changed. Furthermore some key combination might have a special meaning for the OS and should therefore not be used.

General - Themes



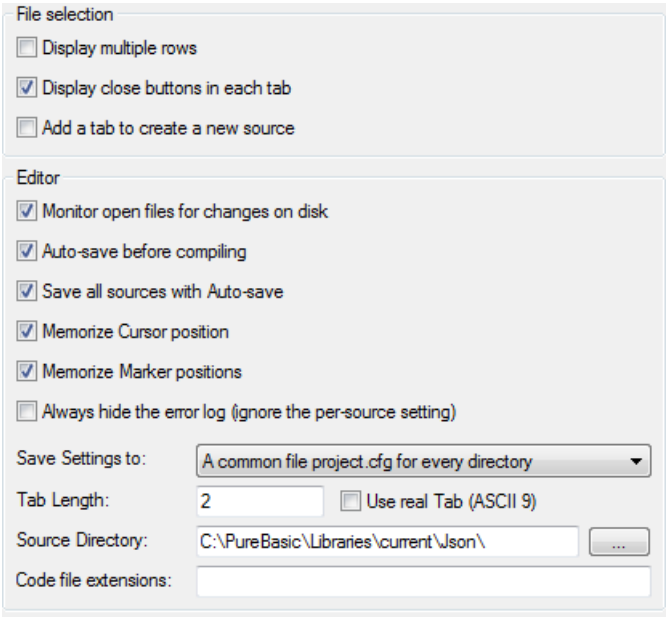
This section shows the available icon themes for the IDE and allows to select the theme to use. The IDE comes with two themes by default.
More themes can be easily added by creating a zip-file containing the images (in png format) and a "Theme.prefs" file to describe the theme. The zip-file has to be copied to the "Themes" folder in the PureBasic installation directory to be recognized by the IDE. The "SilkTheme.zip" file can be used as an example to create a new theme.
Display Icons in the Menu
Allows to hide/show the images in the IDE menus.
Show main Toolbar
Allows to hide/show the main toolbar in order to gain space for the editing area.

General - Toolbar



This allows to fully customize the main Toolbar. By selecting an entry and using the Buttons in the "Position" section, you can change the order. The "Item Settings" section can be used to modify the entry or add a new one. New ones are always added at the end of the list.
Types of items:
Separator : a vertical separator line.
Space : an empty space, the size of one toolbar icon.
Standard Icon : allows you to select a OS standard icon from the combo box on the right.
IDE Icon : allows you to select one of the IDE's own icons in the combo box on the right.
Icon File : allows you to specify your own icon file to use in the edit box on the right (PNG files are supported on all platforms, Windows additionally supports icon files).
If you do not select a separator or space, you can specify an action to carry out when the button is pressed:
Menu Item : carries out the menu command specified in the combo box on the right.
Run tool : executes the external tool specified in the combo box on the right.
The "Default Sets" section contains two standard toolbar sets which you can select, and later modify.

Editor



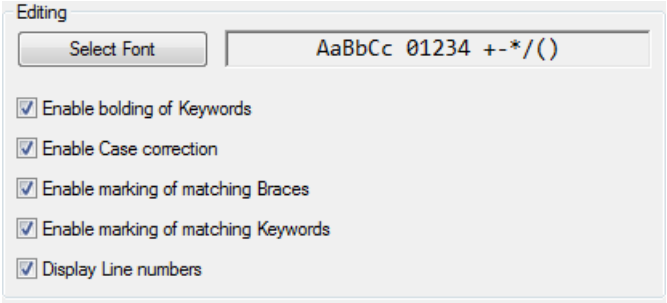
Settings that affect the management of the source codes.
Monitor open files for changes on disk
Monitors all open files for changes that are made to the files on disk while they are edited in the IDE. If modifications are made by other programs, a warning is displayed with the choice to reload the file from disk.
Auto-save before compiling
Saves the current source code before each compile/run or executable creation. Note that any open include files are not automatically saved.
Save all sources with Auto-save
Saves all sources instead of just the current one with one of the Auto-save options.
Memorize cursor position
Saves the current cursor position, as well as the state of all folding marks with the compiler options for the source file.
Memorize Marker positions
Saves all the Markers with the options for the source file.
Always hide the error log
The error log can be shown/hidden on a per-source basis. This option provides a global setting to ignore the per-source setting and never display the error log. It also removes the corresponding menu entries from the IDE menu.
Save settings to
This option allows to specify where the compiler options of a source file are saved:

The end of the Source file
Saves the settings as a special comment block at the end of each source file.
The file <filename>.pb.cfg
Creates a .pb.cfg file for each saved source code that contains this information.
A common file project.cfg for every directory
Creates a file called project.cfg in each directory where PB files are saved. This one file will contain the options for all files in that directory.
Don't save anything
No options are saved. When reopening a source file, the defaults will always be used.

Tab Length
Allows to specify how many spaces are inserted each time you press the Tab key.
Use real Tab (Ascii 9)
If set, the tab key inserts a real tab character instead of spaces. If not set, there are spaces inserted when Tab is pressed.

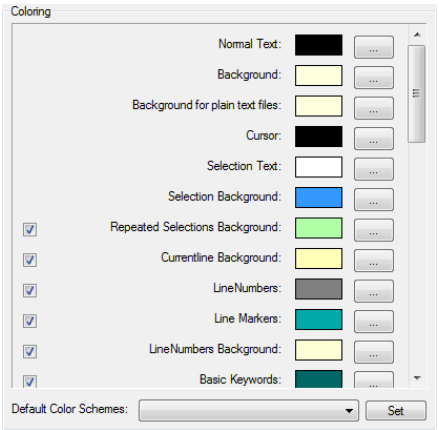
Note that if real tab is used, the "Tab Length" option specifies the size of one displayed tab character.
Source Directory
Specifies the default directory used in the Open and Save dialogs if no other files are currently open (if another file is open, its path will be used as default).
Set this to the path were you usually save the source codes.
Code file extensions
The IDE detects code files by their extension (pb, pbi or pbf by default). Non-code files are edited in a "plain text" mode in which code-related features are disabled. This setting causes the IDE to recognize further file extensions as code files. The field can contain a comma-separated list (i.e. "pbx, xyz") of extensions to recognize.

Editor - Editing



Use "Select Font" to change the font used to display the source code. To ensure a good view of the source code, it should be a fixed-size font, and possibly even one where bold characters have the same size as non-bold ones.
Enable bolding of keywords
If your font does not display bold characters in the same size as non-bold ones, you should disable this option. If disabled, the keywords will not be shown as bold.
Enable case correction
If enabled, the case of PureBasic keywords, PureBasic Functions as well as predefined constants will automatically be corrected while you type.
Enable marking of matching Braces
If enabled, the brace matching the one under the cursor will be highlighted.
Enable marking of matching Keywords
If enabled, the keyword(s) matching the one under the cursor will be underlined.
Display line numbers
Shows or hides the line number column on the left.

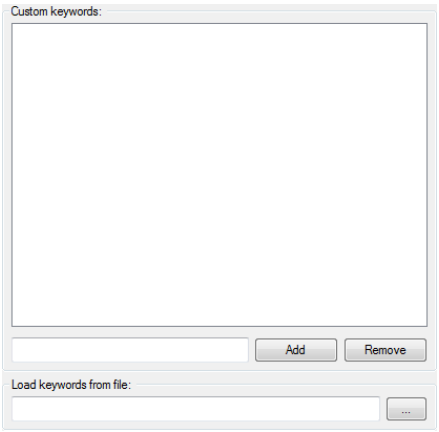
Editor - Coloring



Here you can change the color settings for the syntax coloring, as well as the debugger marks. Default

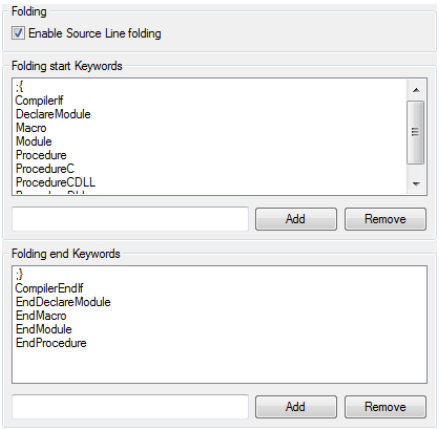
color schemes can be selected from the box on the bottom, and also modified after they have been set. Individual color settings can be disabled by use of the checkboxes.
Note: The 'Accessibility' color scheme has (apart from high-contrast colors) a special setting to always use the system color for the selection in the code editor. This helps screen-reader applications to better detect the selected text.
More color schemes can be added by creating scheme definition files with the name format "SchemeName.prefs". The files have to be copied to the "ColorSchemes" folder in the PureBasic installation directory to be recognized by the IDE. The default scheme files can be used as examples to create a new scheme, or you can export your colors from the Import/Export section of the Preferences window (select "Include Color settings").

Editor - Coloring - Custom Keywords



In this section, a list of custom keywords can be defined. These keywords can have a special color assigned to them in the coloring options and the IDE will apply case-correction to them if this feature is enabled. This allows for applying a special color to special keywords by preprocessor tools or macro sets, or to simply have some PB keywords colored differently.
Note that these keywords take precedence above all other coloring in the IDE, so this allows to change the color or case correction even for PureBasic keywords.
The keywords can be either entered directly in the preferences or specified in a text file with one keyword per line (or both).

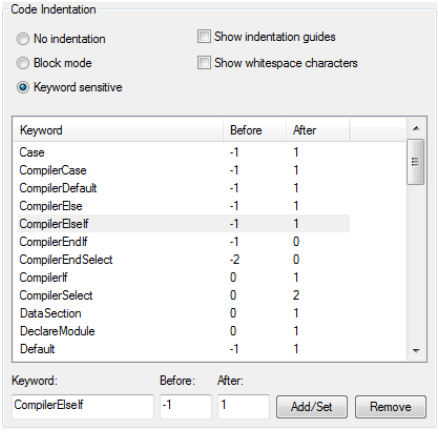
Editor - Folding



Here you can set the keywords in the source code that start/end a foldable section of code. You can add any number of words that will mark such a sections. You can also choose to completely disable the folding feature.
Words that are found inside comments are ignored, unless the defined keyword includes the comment

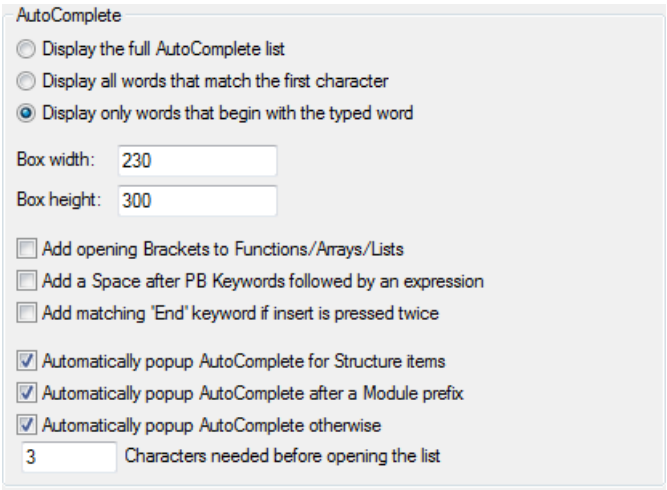
symbol at the start (like the default ";{" keyword).
A keyword may not include spaces.

Editor - Indentation



Here you can specify how the editor handles code indentation when the return key is pressed.
No indentation
Pressing return always places the cursor at the beginning of the next line.
Block mode
The newly created line gets the same indentation as the one before it.
Keyword sensitive
Pressing the return key corrects the indentation of both the old line and the new line depending on the keywords on these lines. The rules for this are specified in the keyword list below. These rules also apply when the "Reformat indentation" item in the edit menu is used.
Show indentation guides
Causes vertical lines to be shown to visualize the indentation on each line. This makes it easier to see which source lines are on the same level of indentation.
Show whitespace characters
Causes whitespace characters to be visible as little dots (spaces) or arrows (tab characters).
The keyword list contains the keywords that have an effect on the indentation. The "Before" setting specifies the change in indentation on the line that contains the keyword itself while the "After" setting specifies the change that applies to the line after it.

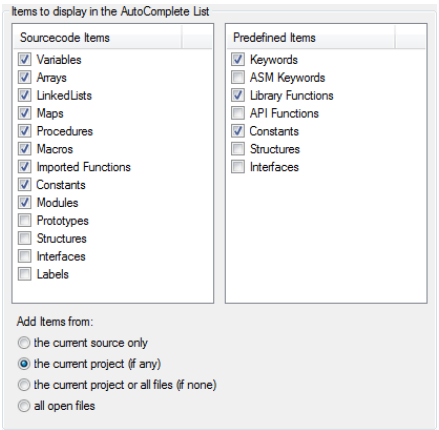
Editor - Auto complete



Display the full Auto complete list
Always displays all keywords in the list, but selects the closest match.

Display all words that start with the first character
Displays only those words that start with the same character as you typed. The closest mach is selected.
Display only words that start with the typed word
Does not display any words that do not start with what you typed. If no words match, the list is not displayed at all.
Box width / Box height
Here you can define the size of the auto complete list (in pixel). Note that these are maximum values. The displayed box may become smaller if there are only a few items to display.
Add opening Brackets to Functions/Arrays/Lists
Will automatically add a "(" after any function/Array/List inserted by auto complete. Functions with no parameters or lists get a "()" added.
Add a Space after PB Keywords followed by an expression
When inserting PB keywords that cannot appear alone, a space is automatically added after them.
Add matching End' keyword if insert shortcut is pressed twice
When the selected keyword has a complementary end keyword, for example "EndSelect" to "Select" or "EndIf" to "If", the insert shortcut (default Tab) can be repeated a second time to automatically add this text too. The end keyword will be inserted after the cursor, so you can continue typing after the first keyword that was inserted.
Automatically popup AutoComplete for Structure items
Displays the list automatically whenever a structured variable or interface is entered and the "\" character is typed after it to show the list of possible structure fields. If disabled, the list can still be displayed by pressing the keyboard shortcut to open the AutoComplete window (usually Ctrl+Space, this can be modified in the Shortcuts section of the Preferences).
Automatically popup AutoComplete outside of Structures
Displays the list automatically when the current word is not a structure after a certain amount of characters has been typed, and a possible match in the list is found. If disabled, the list can still be displayed by pressing the assigned keyboard shortcut.
Characters needed before opening the list
Here you can specify how many characters the word must have minimum before the list is automatically displayed.

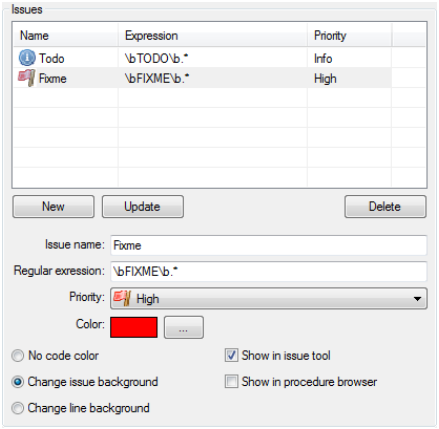
Editor - Auto complete - Displayed items



This shows a list of possible items that can be included with the possible matches in the AutoComplete list.
Source code Items
Items defined in the active source code, or other open sources (see below).
Predefined Items
Items that are predefined by PureBasic, such as the PureBasic keywords, functions or predefined constants.
Add Items from: the current source only
Source code items are only added from the active source code.
Add Items from: the current project (if any)
Source code items are added from the current project if there is one. The other source codes in the

project do not have to be currently open in the IDE for this.
Add Items from: the current project or all files (if none)
Source code items are added from the current project. If the current source code does not belong to the open project then the items from all open source codes will be added.
Add Items from: all open files
Source code items are added from all currently open source codes.

Editor - Issues



Allows to configure the collection of 'issue' markers from comments in the source code. Issue markers can be displayed in the Issues or ProcedureBrowser tool, and they can be marked within the source code with a separate background color.
A definition for an issue consists of the following:

Issue name
A name for the type of issue.
Regular expression
A regular expression defining the pattern for the issue. This regular expression is applied to all comments in the source code. Each match of the expression is considered to match the issue type.
Two optional named groups (named "display" and "mark") can be included in the regex to control which part of the match is highlighted and which part is displayed in the tool.
Example: The default TODO pattern, but only display the part after the TODO in the tool

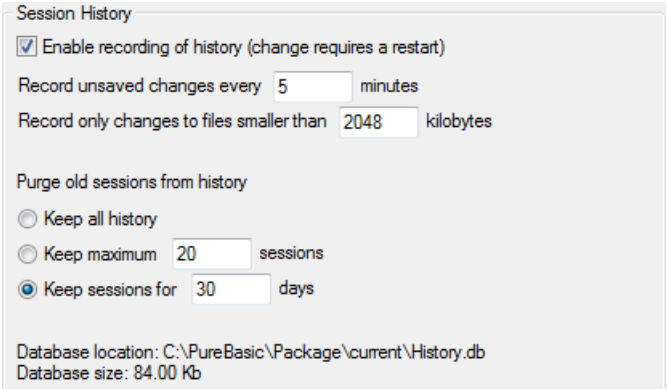
```
\bTODO\b(?<display>.*)
```

Example: Match everything in {}, but only highlight the inner part (display the full match in the tool)

```
{(?<mark>.*)}
```

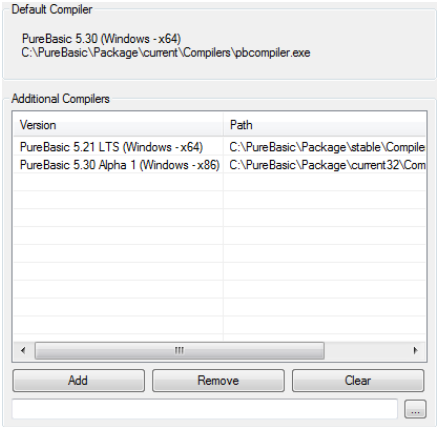
Both named groups can also be used in a single regex for full control. If no groups with these names exist in the regex, the full match will be used.
Priority
Each issue type is assigned a priority. The priority can be used to order and filter the displayed issues in the issue tool.
Color
The color used to mark the issue in the source code (if enabled). The color will be used to mark the background of either only the issue text itself, or the entire code line depending on the coloring option.
Show in issue tool
If enabled, any found issues of this type are listed in the issues tool. This option can be disabled to cause an issue to only be marked in the source code with a special background color if desired.
Show in procedure browser
If enabled, any found issues are shown as an entry in the procedure browser tool.

Editor - Session history



Allows to configure how the session history is recording changes.
Enable recording of history
Enable or disable the history session recording. When enabled, all the changes made to a file will be recorded in the background in a database. A session is created when the IDE launch, and is closed when the IDE quits. This is useful to rollback to a previous version of a file, or to find back a deleted or corrupted file. It's like a very powerful source backup tool, limited in time (by default one month of recording). It's not aimed to replace a real source versioning system like SVN or GIT. It's complementary to have finer change trace. The source code will be stored without encryption, so if you are working on sensitive source code, be sure to have this database file in a secure location, or disable this feature. It's possible to define the session history database location using an IDE command-line switch.
Record change every X minutes
Change the interval between each silent recording (when editing). A file will be automatically recorded when saving or closing it.
Record only changes to files smaller than X kilobytes
Change the maximum size (in kilobytes) of the files being recorded. This allow to exclude very big files which could make the database grow a lot.
Keep all history
Keep all the history, the database is never purged. It will always grows, so it should be watched.
Keep maximum X sessions
After reaching the maximum number of sessions, the oldest session will be removed from the database.
Keep sessions for X days
After reaching the maximum number of days, the session will be removed from the database.

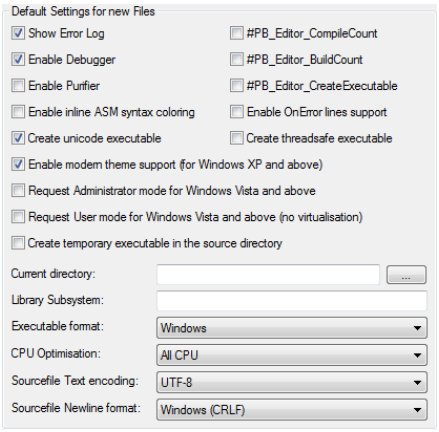
Compiler



This page allows to select additional compilers which should be available for compilation in the Compiler Options . This allows switching between different compilers of the same version (like the x86 and x64 compilers) or even switching between different versions easily.
Any PureBasic compiler starting from Version 4.10 can be added here. The target processor of the

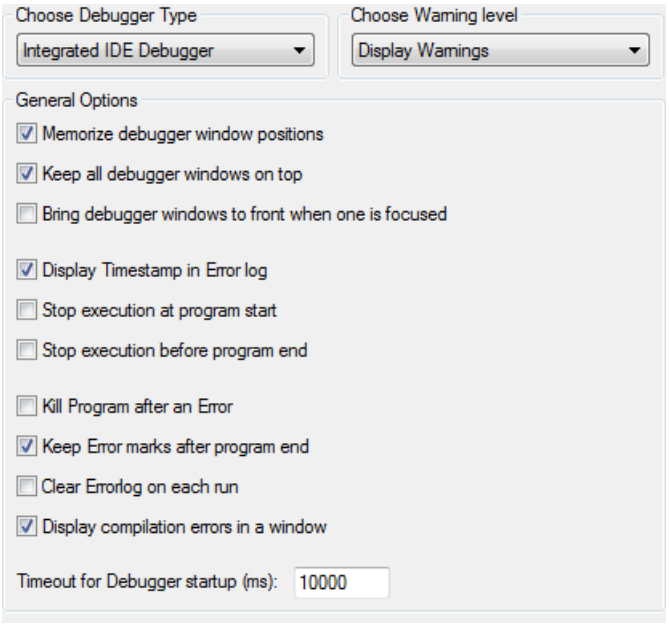
selected compilers does not have to match that of the default compiler, as long as the target operating system is the same. The list displays the compiler version and path of the selected compilers.
The information used by the IDE (for code highlighting, auto complete, structure viewer) always comes from the default compiler. The additional compilers are only used for compilation.

Compiler - Defaults



This page allows setting the default compiler options that will be used when you create a new source code with the IDE.
For an explanation of the meaning of each field, see the Compiler Options .

Debugger



Settings for the internal Debugger, or the Standalone Debugger. The command-line debugger is configured from the command-line only.
Debugger Type
Select the type of debugger you want to use when compiling from the IDE here.
Choose Warning level
Select the action that should be taken if the debugger issues a warning. The available options are:
Ignore Warnings: Warnings will be ignored without displaying anything.
Display Warnings: Warnings will be displayed in the error log and the source code line will be marked, but the program continues to run.

Treat Warnings as Errors: A warning will be treated like an error.
See Using the debugger for more information on debugger warnings and errors.

Memorize debugger window positions
The same as the "Memorize Window positions" for in the General section, but for all Debugger windows.

Keep all debugger windows on top
All debugger windows will be kept on top of all other windows, even from other applications.

Bring Debugger windows to front when one is focused
With this option set, focusing one window that belongs to the debugger of a file, all windows that belong to the same debugging session will be brought to the top.

Display Timestamp in error log
Includes the time of the event in the error log.

Stop execution at program start
Each program will be started in an already halted mode, giving you the opportunity to start moving step-by-step, right from the start of the program.

Stop execution before program end
Stops the program execution right before the executable would unload. This gives you a last chance to use the debugging tools to examine Variables or Memory before the program ends.

Kill Program after an Error
If a program encounters an error, it will be directly ended and all debugger windows closed. This gives the opportunity to directly modify the code again without an explicit "Kill Program", but there is no chance to examine the program after an error.

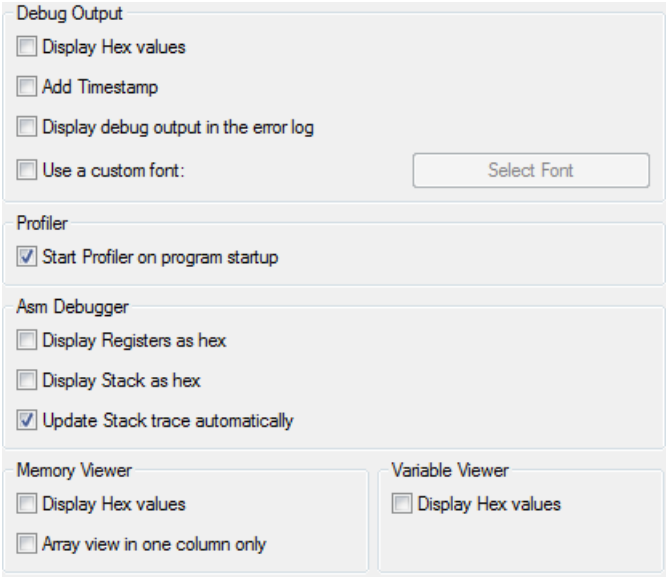
Keep Error marks after program end
Does not remove the lines marked with errors when the program ends. This gives the opportunity to still see where an error occurred while editing the code again.
The marks can be manually removed with the "Clear error marks" command in the "Error log" submenu of the debugger menu.

Clear error log on each run
Clears the log window when you execute a program. This ensures that the log does not grow too big (this option is also available with the command-line debugger selected).

Timeout for Debugger startup
Specifies the time in milliseconds how long the debugger will wait for a program to start up before giving up. This timeout prevents the debugger from locking up indefinitely if the executable cannot start for some reason.

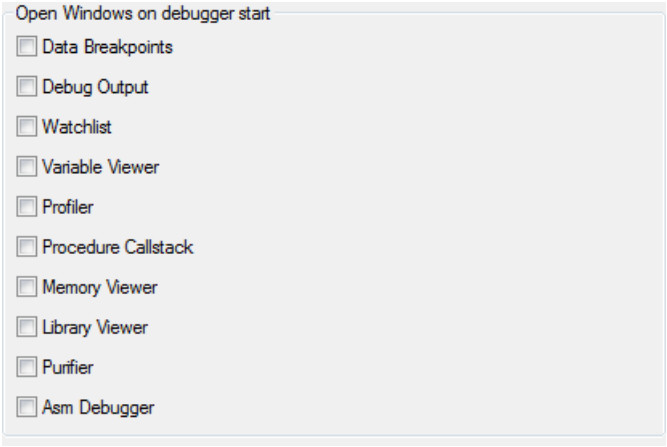
Debugger - Individual Settings

This allows setting options for the individual debugger tools. The "Display Hex values" options switch between displaying Byte, Long and Word as decimal or hexadecimal values.



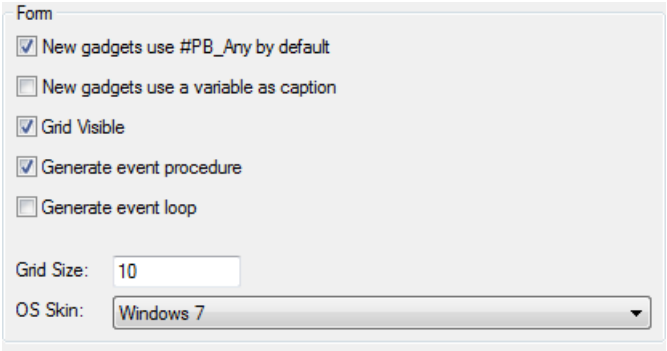
Debug Output Add Timestamp
Adds a timestamp to the output displayed from the Debug command.
Debug Output - Display debug output in the error log
With this option enabled, a Debug command in the code will not open the Debug Output window , but instead show the output in the error log .
Debug Output Use custom font
A custom font can be selected here for the debug output window . This allows to specify a smaller font for much output or a proportional one if this is desired.
Profiler - Start Profiler on program startup
Determines whether the profiler tool should start recording data when the program starts.
ASM Debugger Update Stack trace automatically
Updates the stack trace automatically on each step/stop you do. If disabled, a manual update button will be displayed in the ASM window.
Memory Viewer Array view in one column only
If the memory area is displayed in array view, this option selects whether it should be multi-column (with 16 bytes displayed in each column) or with only one column.

Debugger - Default Windows



The debugger tools you select on this page will automatically be opened each time you run a program with enabled debugger.

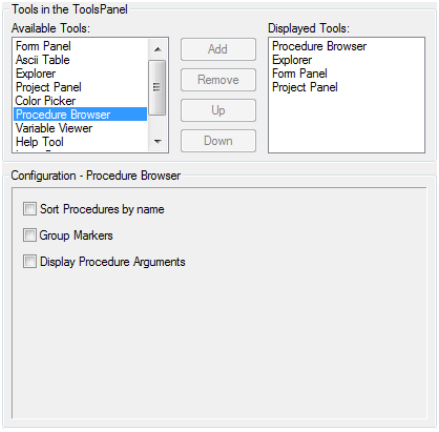
Form



Allows to customize the integrated form designer behavior.
New gadgets use #PB_Any by default
If enabled, new gadget creation will use #PB_Any instead of static enumeration numbering.
New gadgets use a variable as caption
If enabled, new gadget will use a variable instead of static text as caption. It can be useful to easily localize the form.

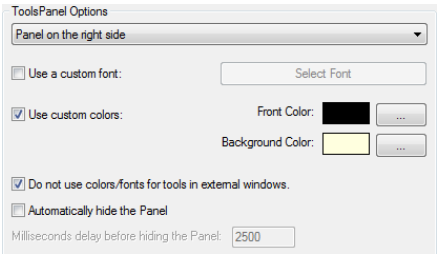
Grid visible
If enabled, the grid will be visible on the form designer, to ease gadget alignment.
Generate event procedure
If enabled, an event procedure will be automatically generated (and updated).
Generate event loop
If enabled, a basic event loop will be generated for the form.
Grid size
Space between two grid points, in pixels.
OS skin
Skin to use for the form designer.

Tools Panel



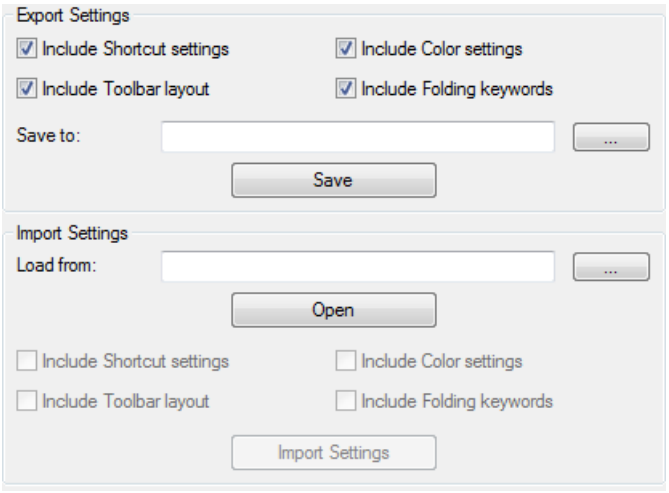
This allows configuring the internal tools that can be displayed in the side panel. Each tool that is in the ”Displayed Tools” list is displayed in the Panel on the side of the edit area. Each tool that is not listed there is accessible from the Tools menu as a separate window.
Put only those tools in the side panel that you use very frequently, and put the most frequently used first, as it will be the active one once you open the IDE.
By selecting a tool in either of the lists, you get more configuration options for that tool (if there are any) in the ”Configuration” section below.
Here is an explanation of those tools that have special options:
Explorer
You can select between a Tree or List display of the file-system. You can also set whether the last displayed directory should be remembered, or if the Source Path should be the default directory when the IDE is started.
Procedure Browser
”Sort Procedures by Name” : sorts the list alphabetically (by default they are listed as they appear in the code).
”Group Markers” : groups the ”;-” markers together.
”Display Procedure Arguments” : Displays the full declaration of each procedure in the list.
Variable Viewer
The ”Display Elements from all open sources” option determines whether the list should only include items from this code, or from all open codes.
Furthermore you can select the type of items displayed in the Variable viewer.
Help Tool
”Open sidebar help on F1”: specifies whether to open the help tool instead of the separate help viewer when F1 is pressed.

Tools panel - Options



Here you can customize the appearance of the Tools Panel a bit more. You can select the side on which it will be displayed, a Font for its text, as well as a foreground and background color for the displayed tools. The font and color options can be disabled to use the OS defaults instead.
Do not use colors/fonts for tools in external windows
If set, the color/font options only apply to the Tools displayed in the Panel, those that you open from the Tools menu will have the default colors.
Automatically hide the Panel
To save space, the Panel will be hidden if the mouse is not over it. Moving the mouse to the side of the IDE will show it again.
Milliseconds delay before hiding the Panel
Sets a timeout in ms, after which the Panel is hidden if you leave it with the mouse.

Import/Export



This section allows you to export the layout settings of the IDE in a platform independent format, which allows you to import them again into the PureBasic IDE on another Operating System, or to share your options with other PB users.
To export your settings, select what types of settings you want to include, select a filename and press the "Save" button.
To import settings, select the filename and press "Open". You will then see the options that are included in this file as enabled checkboxes. After selecting what you want to import, click the "Import Settings" button.
For the new settings to take effect, you have to first click the apply button.
Note: You can import the style files from the jaPBe Editor, but only the color settings will be imported.

Chapter 19

Command-line options for the IDE

The PureBasic IDE allows you to modify the paths and files being used from the command-line. This allows you to create several shortcuts that start the IDE with different configurations for different users, or for different projects.
There are also options for compiling PureBasic projects directly from the command-line. Building a project from the command-line involves the same actions like at choosing the 'Build Target' or 'Build all Targets' from the compiler menu .
General options:

```
/VERSION                  displays the IDE version and exits
/HELP or /?               displays a description of the command-line
 arguments
```

Options for launching the IDE:

```
/P <Preferences file>     loads/saves all the configuration to/from
 the given file
/T <Templates file>       loads/saves the code templates from/to the
 given file
/A <tools file>           loads/saves the configuration of the
 external tool from/to this file
/S <Source path>          overwrites the "Source path" setting from
 the preferences
/E <Explorer path>        starts the Explorer tool with the given path
/L <Line number>          moves the cursor to the given line number in
 the last opened file
/H <HistoryDatabase>      specify the file to use for the session
 history database
/NOEXT                    disables the registering of the .pb
 extension in the registry
/LOCAL                    puts all preferences in the PureBasic
 directory instead of the user profile location
/PORTABLE                 the same as /LOCAL and /NOEXT combined
```

Options for building projects:

```
/BUILD <file>             specifies the project file to build
/TARGET <target>          specifies the target to build (the default
 is to build all targets)
/QUIET                    hides all build messages except errors
/READONLY                 does not update the project file after
 compiling (with new access time and build counters)
```

The default files for /P /T and /A are saved in the %APPDATA%\PureBasic\ directory on the system.

The /NOEXT command is useful when you have several different PB versions at once (for testing of beta versions for example), but want the .pb extension to be associated with only one of them. The /PORTABLE command can be used to keep all configuration inside the local directory to easily copy PureBasic to different computers (or run it from USB sticks for example).
Example:

```
1   PureBasic.exe Example.pb /PORTABLE
```

You can also put the filenames of source files to load on the command-line. You can even specify wildcards for them (so with "*.pb" you can load a whole directory).

Part III

Language Reference

Chapter 20

Working with different number bases

(Note: These examples use the ^ symbol to mean 'raised to the power of' - this is only for convenience in this document, PureBasic does not currently have a power-of operator! Use the PureBasic command Pow() from the "Math" Library instead.)

Introduction

A number base is a way of representing numbers, using a certain amount of possible symbols per digit. The most common you will know is base 10 (a.k.a. decimal), because there are 10 digits used (0 to 9), and is the one most humans can deal with most easily.
The purpose of this page is to explain different number bases and how you can work with them. The reason for this is that computers work in binary (base 2) and there are some cases where it is advantageous to understand this (for example, when using logical operators, bit masks, etc).

Overview of number bases

Decimal System

Think of a decimal number, and then think of it split into columns. We'll take the example number 1234. Splitting this into columns gives us:

```
1    2    3    4
```

Now think of what the headings for each column are. We know they are units, tens, hundreds and thousands, laid out like this:

```
1000   100    10     1
   1     2     3     4
```

We can see that the number 1234 is made up out of

```
   1*1000=1000
+  2*  100=  200
+  3*   10=   30
+  4*    1=    4
Total    =1234
```

If we also look at the column headings, you'll see that each time you move one column to the left we multiply by 10, which just so happens to be the number base. Each time you move one column to the right, you divide by 10. The headings of the columns can be called the weights, since to get the total

number we need to multiply the digits in each column by the weight. We can express the weights using indices. For example 10 ^2 means '10 raised to the power of two' or 1*10*10 (=100). Similarly, 10 ^4 means 1*10*10*10*10 (=10000). Notice the pattern that whatever the index value is, is how many times we multiply the number being raised. 10 ^0 means 1 (since we multiply by 10 no times). Using negative numbers shows that we need to divide, so for example 10 ^-2 means 1/10/10 (=0.01). The index values make more sense when we give each column a number - you'll often see things like 'bit 0' which really means 'binary digit in column 0'.

```
In this example, ^ means raised to the power of, so 10^2 means 10
 raised to the power of 2.
Column number              3     2     1     0
Weight (index type)     10^3  10^2  10^1  10^0
Weight (actual value)   1000   100    10     1
Example number (1234)      1     2     3     4
```

A few sections ago we showed how to convert the number 1234 into its decimal equivalent. A bit pointless, since it was already in decimal but the general method can be shown from it - so this is how to convert from any number to its decimal value:

```
B = Value of number base

1) Separate the number in whatever base you have into it's columns.
 For
   example, if we had the value 'abcde' in our fictional number base
'B',
   the columns would be:   a   b   c   d   e

2) Multiply each symbol by the weight for that column (the weight
 being
   calculated by 'B' raised to the power of the column number):
     a * B^4 = a * B * B * B * B
     b * B^3 = b * B * B * B
     c * B^2 = c * B * B
     d * B^1 = d * B
     e * B^0 = e

3) Calculate the total of all those values. By writing all those
 values in their
   decimal equivalent during the calculations, it becomes far easier
 to see the
   result and do the calculation (if we are converting into decimal).
```

Converting in the opposite direction (from decimal to the number base 'B') is done using division instead of multiplication:

```
1) Start with the decimal number you want to convert (e.g. 1234).

2) Divide by the target number base ('B') and keep note of the result
 and the
   remainder.

3) Divide the result of (2) by the target number base ('B') and keep
 note of
   the result and the remainder.

4) Keep dividing like this until you end up with a result of 0.

5) Your number in the target number base is the remainders written in
 the order
   most recently calculated to least recent. For example, your number
 would be
```

```
    the remainders of the steps in this order 432.
```

More specific examples will be given in the sections about the specific number bases.

Binary System

Everything in a computer is stored in binary (base 2, so giving symbols of '0' or '1') but working with binary numbers follows the same rules as decimal. Each symbol in a binary number is called a bit, short for binary digit. Generally, you will be working with bytes (8-bit), words (16-bit) or longwords (32-bit) as these are the default sizes of PureBasic's built in types. The weights for a byte are shown:

```
(^ means 'raised to the power of', number base is 2 for binary)
Bit/column number          7     6     5     4     3     2     1     0
Weight (index)           2^7   2^6   2^5   2^4   2^3   2^2   2^1   2^0
Weight (actual value)    128    64    32    16     8     4     2     1
```

So, for example, if we had the number 00110011 (Base 2), we could work out the value of it like this:

```
  0 * 128
+ 0 *  64
+ 1 *  32
+ 1 *  16
+ 0 *   8
+ 0 *   4
+ 1 *   2
+ 1 *   1
=      51
```

An example of converting back would be if we wanted to write the value 69 in binary. We would do it like this:

```
69 / 2 = 34 r 1      ^
34 / 2 = 17 r 0     /|\
17 / 2 =  8 r 1      |
 8 / 2 =  4 r 0      |     Read remainders in this direction
 4 / 2 =  2 r 0      |
 2 / 2 =  1 r 0      |
 1 / 2 =  0 r 1      |
(Stop here since the result of the last divide was 0)

Read the remainders backwards to get the value in binary = 1000101
```

Another thing to consider when working with binary numbers is the representation of negative numbers. In everyday use, we would simply put a negative symbol in front of the decimal number. We cannot do this in binary, but there is a way (after all, PureBasic works mainly with signed numbers, and so must be able to handle negative numbers). This method is called 'twos complement' and apart from all the good features this method has (and won't be explained here, to save some confusion) the simplest way to think of it is the weight of the most significant bit (the MSb is the bit number with the highest weight, in the case of a byte, it would be bit 7) is actually a negative value. So for a two's complement system, the bit weights are changed to:

```
(^ means 'raised to the power of', number base is 2 for binary)
Bit/column number          7     6     5     4     3     2     1     0
Weight (index)          -2^7   2^6   2^5   2^4   2^3   2^2   2^1   2^0
Weight (actual value)   -128    64    32    16     8     4     2     1
```

and you would do the conversion from binary to decimal in exactly the same way as above, but using the new set of weights. So, for example, the number 10000000 (Base 2) is -128, and 10101010 (Base 2) is -86.

To convert from a positive binary number to a negative binary number and vice versa, you invert all the bits and then add 1. For example, 00100010 would be made negative by inverting -> 11011101 and adding 1 -> 11011110.
This makes converting from decimal to binary easier, as you can convert negative decimal numbers as their positive equivalents (using the method shown above) and then make the binary number negative at the end.
Binary numbers are written in PureBasic with a percent symbol in front of them, and obviously all the bits in the number must be a '0' or a '1'. For example, you could use the value %110011 in PureBasic to mean 51. Note that you do not need to put in the leading '0's (that number would really be %00110011) but it might help the readability of your source if you put in the full amount of bits.

Hexadecimal System

Hexadecimal (for base 16, symbols '0'-'9' then 'A'-'F') is a number base which is most commonly used when dealing with computers, as it is probably the easiest of the non-base 10 number bases for humans to understand, and you do not end up with long strings of symbols for your numbers (as you get when working in binary).
Hexadecimal mathematics follows the same rules as with decimal, although you now have 16 symbols per column until you require a carry/borrow. Conversion between hexadecimal and decimal follows the same patterns as between binary and decimal, except the weights are in multiples of 16 and divides are done by 16 instead of 2:

```
Column number              3      2      1      0
Weight (index)          16^3   16^2   16^1   16^0
Weight (actual value)   4096    256     16      1
```

Converting the hexadecimal value BEEF (Base 16) to decimal would be done like this:

```
  B * 4096 = 11 * 4096
+ E *  256 = 14 *  256
+ E *   16 = 14 *   16
+ F *    1 = 15 *    1
=                48879
```

And converting the value 666 to hexadecimal would be done like this:

```
666 / 16 = 41 r 10    ^
 41 / 16 =  2 r  9   /|\    Read digits in this direction,
 remembering to convert
  2 / 16 =  0 r  2    |     to hex digits where required
(Stop here, as result is 0)
Hexadecimal value of 666 is 29A
```

The really good thing about hexadecimal is that it also allows you to convert to binary very easily. Each hexadecimal digit represents 4 bits, so to convert between hexadecimal and binary, you just need to convert each hex digit to 4 bits or every 4 bits to one hex digit (and remembering that 4 is an equal divisor of all the common lengths of binary numbers in a CPU). For example:

```
Hex number          5        9        D        F          4E
Binary value     0101     1001     1101     1111    01001110
```

When using hexadecimal values in PureBasic, you put a dollar sign in front of the number, for example $BEEF.

Chapter 21

Break : Continue

Syntax

```
Break [Level]
```

Description

Break provides the ability to exit during any iteration, for the following loops: Repeat , For , ForEach and While . The optional 'level' parameter specifies the number of loops to exit from, given several nested loops. If no parameter is given, Break exits from the current loop.

Example: Simple loop

```
1  For k=0 To 10
2    If k=5
3      Break  ; Will exit directly from the For/Next loop
4    EndIf
5    Debug k
6  Next
```

Example: Nested loops

```
1  For k=0 To 10
2    Counter = 0
3    Repeat
4      If k=5
5        Break 2 ; Will exit directly from the Repeat/Until and For/Next
       loops
6      EndIf
7      Counter+1
8    Until Counter > 1
9    Debug k
10 Next
```

Syntax

```
Continue
```

Description

Continue provides the ability to skip straight to the end of the current iteration (bypassing any code in between) in the following loops: Repeat , For , ForEach , and While .

Example

```
1  For k=0 To 10
2    If k=5
3      Continue  ; Will skip the 'Debug 5' and go directly to the next
     iteration
4    EndIf
5    Debug k
6  Next
```

Chapter 22

Using the command line compiler

Introduction

The command line compiler is located in the subdirectory 'compilers' from the PureBasic folder. The easier way to access it is to add this directory in the PATH variable, which will give access to all the commands of this directory at all times.
There is two compilers for PureBasic:
- 'pbcompiler' which is generating assembly code (x86 and x64)
- 'pbcompilerc' (or 'pbcompiler' on platform which doesn't support the assembly backend) which is generating C code
The C backend compiler is available on all supported platforms, while the ASM backend compiler is only supported on Windows (x86, x64), Linux (x86, x64) and OS X (x64).
The backend C compiler does not allow entering assembly code.
On Windows a 'PureBasic-CLI.cmd' file is available in the PureBasic root directly to easily open a commandline prompt to use the PureBasic commandline tools.

Cross-platform command switches

Note: All switches name which begin with '/' are only valid on Windows (ie: '/DEBUGGER'). If you need to write cross-plaform commandline, you can use the '-' or '–' format (ie: '-d' or '–debugger).
-h, –help, /?: displays a quick help about the compiler.
-d, –debugger, /DEBUGGER: enables the debugger support.
-cl, –console, /CONSOLE: output file in Console format. Default format on Windows is Win32.
-pf, –purifier, /PURIFIER: enables the purifier. The debugger needs to be activated to have any effect.
-o, –output, /OUTPUT "filename": creates a standalone executable or DLL specified by the filename at the desired path location. On MacOS X it's possible to create an application bundle by adding '.app' to the name of the executable. This way the whole directory structure will be automatically created.
-r, –resident, /RESIDENT "filename": creates a resident file specified by the filename.
-i, –import, /IMPORT "filename": creates an import file to the given filename. Only one Import/EndImport block is allowed in the file. The imported functions will be loaded automatically for every PureBasic projects.
-l or –linenumbering, /LINENUMBERING: adds lines and files information to the executable, which can slow it considerably. This allows the OnError library to report the file and line-number of an error.
-q, –quiet, /QUIET: disables all unnecessary text output, very useful when using another editor.
-sb, –standby, /STANDBY: loads the compiler in memory and wait for external commands (editor, scripts...). More information about using this flag is available in the file 'CompilerInterface.txt' from the PureBasic 'SDK' directory.
-ir, –ignoreresident, /IGNORERESIDENT "filename": doesn't load the specified resident file when the compiler starts. It's mostly useful when updating a resident which already exists, so it won't load it.
-co, –constant, /CONSTANT name=value: creates the specified constant with the given expression. Example: 'pbcompiler test.pb /CONSTANT MyConstant=10'. The constant #MyConstant will be

created on the fly with a value of 10 before the program gets compiled.
-t, –thread, /THREAD: uses thread safe runtime for strings and general routines.
-s, –subsystem, /SUBSYSTEM "name": uses the specific subsystem to replace a set of internal functions. The subsystem name is not case-sensitive. For more information, see subsystems .
-k, –check, /CHECK: checks the syntax only, doesn't create or launch the executable.
-z, –optimizer, /OPTIMIZER: enable the code optimizer.
-c, –commented, /COMMENTED: creates a commented 'purebasic.asm' or 'purebasic.c' output file when creating an executable. This file can be re-assembled with the '–reasm' option later when the needed modifications have been made. This option is for advanced programmers only.
-ra, –reasm, /REASM: re-assembles the 'purebasic.asm' or 'purebasic.c' file into an executable. This allow to use the '–commented' option, modify the generated output and creates the executable again. This option is for advanced programmers only.
-pp, –preprocess, /PREPROCESS "filename": preprocess the source code and write the output in the specified "Filename". The processed file is a single file with all macro expanded, compiler directive handled and multiline resolved. This allows an easier parsing of a PureBasic source file, as all is expanded and is available in a flat file format. No comments are included by default, but the flag '–commented' can be used to have all untouched original source as comments and allow comments processing. The preprocessed file can be recompiled as any other PureBasic source file to create the final executable.
-ls, –liststructures, /LISTSTRUCTURES: creates a file with all the built-in structures known by the compiler. The filename can be specified with '–output'.
-lf, –listfunction, /LISTFUNCTIONS: creates a file with all the built-in purebasic commands known by the compiler. The filename can be specified with '–output'.
-li, –listinterfaces, /LISTINTERFACES: creates a file with all the built-in interfaces known by the compiler. The filename can be specified with '–output'.
-qs, –querystructure, /QUERYSTRUCTURE "name": creates a file with the structure definition. The filename can be specified with '–output.
-g, –language, /LANGUAGE \"language\": uses the specified language for the compiler error messages.
-v, –version, /VERSION: displays the compiler version.
Examples:

```
CLI> pbcompiler sourcecode.pb
```

The compiler will compile the source code and execute it.

```
CLI> pbcompiler sourcecode.pb --debugger
```

The compiler will compile the source code and execute it with debugger.

Windows specific command switches

/ICON "IconName.ico" : add the specified icon to the executable.
/DLL: output file is a DLL .
/XP: Add the Windows theme support to the executable.
/MMX, /3DNOW, /SSE or /SSE2: Creates a processor specific executable. This means if a processor specific routine is available it will be used for this executable. Therefore, the executable will run correctly only on computer with this kind of CPU.
/DYNAMICCPU: Creates a executable containing all the available processor specific routines. When the program is started, it looks for the processor type and then select the more appropriates routines to use. This makes a bigger executable, but result in the fastest possible program.
/RESOURCE "Filename": Add a Windows resource file (.rc) to the created executable or DLL. This should be not a compiled resource file, but an ascii file with the directives in it. It's possible to specify only one resource, but this file can include other resource script if needed.
/LINKER "ResponseFile": Specify a file which contains commands to be passed directly to the linker. On Windows, PureBasic use the PellesC linker (polink), more information about the possible options can be found in the related documentation.
The following two compiler options are needed for creating programs running on Microsoft Vista OS or above (Windows 7/8/10). They are both options for the included manifest, so they are ignored on older windows versions. With none of these switches, the exe will run as a normal user as well, but with virtualization turned on (i.e. registry and file redirection). It is recommended to use the /USER switch

to turn of virtualization for all programs that comply to the standard user privileges, as it is only intended for compatibility for older programs. These options are also available in the IDE compiler options .
/ADMINISTRATOR: Will cause the program to request admin privileges at start. The program will not run without it. This option is needed. Note: You can also debug programs with this flag, but only with the standalone gui debugger (as it must run in elevated mode as well).
/USER: The program will run as the user who started it. Virtualization for the exe is turned off.
/DPIAWARE: Enable DPI support to the executable.
/DLLPROTECTION: Enable DLL preloading protection to the executable. It prevents that system DLLs are first searched for in the program directory instead of in the System32 directory of the Windows operating system.
/UCRT: Use the dynamic universal CRT when creating the executable or DLL. It is recommended when creating an executable or a DLL which doesn't need to support older Windows version below Windows 10. The resulting executable or DLL will be smaller and will get automatic security fixes from the CRT when Microsoft updates it. This option is always enabled on PureBasic for Windows arm64.
Examples:

```
CLI> pbcompiler "C:\Project\Source\DLLSource.pb" /EXE
 "C:\Project\project.dll" /DLL
```

The compiler will compile the source code (here with full path) and create the DLL "project.dll" in the given directory.

Linux specific command switches

-so or –sharedobject "filename": Create a dynamic library (shared object).
-mmx, -3dnow, -sse or -sse2: Creates a processor specific executable. This means if a processor specific routine is available it will be used for this executable. Therefore, the executable will run correctly only on computer with this kind of CPU.
-dc or –dynamiccpu: Creates a executable containing all the available processor specific routines. When the program is started, it looks for the processor type and then select the more appropriates routines to use. This makes a bigger executable, but result in the fastest possible program.

OS X specific command switches

-n or –icon "filename.icns" : add the specified icon to the executable.
-dl or –dylib "filename": Create a dynamic library (dylib object).
-f or –front: put the launched process to front.
-ibp or –ignorebundlepath: don't use the bundle path as current directory.

Chapter 23

Compiler Directives

Syntax

```
CompilerIf <constant expression>
  ...
[CompilerElseIf]
  ...
[CompilerElse]
  ...
CompilerEndIf
```

Description

If the <constant expression> result is true, the code inside the CompilerIf will be compiled, else it will be totally ignored. It's useful when building multi-OSes programs to customize some programs part by using OS specific functions. The And and Or Keywords can be used in <constant expression> to combine multiple conditions.

Example

```
1  CompilerIf #PB_Compiler_OS = #PB_OS_Linux And #PB_Compiler_Processor
     = #PB_Processor_x86
2    ; some Linux and x86 specific code.
3  CompilerEndIf
```

Syntax

```
CompilerSelect <numeric constant>
  CompilerCase <numeric constant>
    ...
  [CompilerDefault]
    ...
CompilerEndSelect
```

Description

Works like a regular Select : EndSelect except that only one numeric value is allowed per case. It will tell the compiler which code should be compiled. It's useful when building multi-OSes programs to customize some programs part by using OS specific functions.

Example

```
CompilerSelect #PB_Compiler_OS
  CompilerCase #PB_OS_MacOS
    ; some Mac OS X specific code
  CompilerCase #PB_OS_Linux
    ; some Linux specific code
CompilerEndSelect
```

Syntax

```
CompilerError <string constant>
CompilerWarning <string constant>
```

Description

Generates an error or a warning, as if it was a syntax error and display the associated message. It can be useful when doing specialized routines, or to inform a source code is not available on an particular OS. Note: CompilerWarning displays warnings, but the compilation process continues, while the command CompilerError stops the compilation process.

Example: Generates an error

```
CompilerIf #PB_Compiler_OS = #PB_OS_Linux
  CompilerError "Linux isn't supported, sorry."
CompilerElse
  CompilerError "OS supported, you can now comment me."
CompilerEndIf
```

Example: Generates a warning

```
CompilerIf #PB_Compiler_OS = #PB_OS_Linux
  CompilerWarning "Linux isn't supported, sorry."
CompilerElse
  CompilerWarning "OS supported, you can now comment me."
CompilerEndIf
```

Syntax

```
EnableExplicit
DisableExplicit
```

Description

Enables or disables the explicit mode. When enabled, all the variables which are not explicitly declared with Define , Global , Protected or Static are not accepted and the compiler will raise an error. It can help to catch typo bugs.

Example

```
EnableExplicit

Define a

a = 20 ; Ok, as declared with 'Define'
b = 10 ; Will raise an error here
```

Syntax

```
EnableASM
DisableASM
```

Description

Enables or disables the inline assembler. When enabled, all the assembler keywords are available directly in the source code, see the inline assembler section for more information.

Example

```
; x86 assembly example
;
Test = 10

EnableASM
  MOV dword [v_Test],20
DisableASM

Debug Test ; Will be 20
```

Syntax

```
DisablePureLibrary <LibraryName>
```

Description

Disables the specified PureLibrary. All its associated commands won't be available when compiling this program. This is especially useful when creating a PureLibrary .

Example

```
DisablePureLibrary Engine3D

InitEngine3D() ; Error, command not found
```

Reserved Constants

The PureBasic compiler has several reserved constants which can be useful to the programmer:

```
 #PB_Compiler_OS : Determines on which OS the compiler is currently
  running. It can be one of the following values:
   #PB_OS_Windows : The compiler is running on Windows
   #PB_OS_Linux   : The compiler is running on Linux
   #PB_OS_MacOS   : The compiler is running on Mac OS X

 #PB_Compiler_Processor : Determines the processor type for which the
  program is created. It can be one of the following:
   #PB_Processor_x86     : x86 processor architecture (also called
  IA-32 or x86-32)
   #PB_Processor_x64     : x86-64 processor architecture (also called
  x64, AMD64 or Intel64)
   #PB_Processor_arm32   : arm32 processor architecture
   #PB_Processor_arm64   : arm64 processor architecture (also called
  M1 on Apple computers)

 #PB_Compiler_Backend : Determines which kind of compiler is currently
  used. It can be one of the following:
   #PB_Backend_Asm    : The compiler generating assembly code is used.
   #PB_Backend_C      : The compiler generating C code is used.

 #PB_Compiler_ExecutableFormat : Determines executable format. It can
  be one of the following:
   #PB_Compiler_Executable : Regular executable
   #PB_Compiler_Console    : Console executable (have an effect only
  on Windows, other act like a regular executable)
   #PB_Compiler_DLL        : Shared DLL (dynlib on MacOS X and shared
  object on Linux)

 #PB_Compiler_Date     : Current date, at the compile time, in the
  PureBasic  date
format (local time).
 #PB_Compiler_File     : Full path and name of the file being
  compiled, useful for debug
purpose.
 #PB_Compiler_FilePath : Full path of the file being compiled, useful
  for debug
purpose.
 #PB_Compiler_Filename : Filename (without path) of the file being
  compiled, useful for debug
purpose.
 #PB_Compiler_Line     : Line number of the file being compiled,
  useful for debug
purpose.
 #PB_Compiler_Procedure: Current procedure name, if the line is inside
  a procedure
.
 #PB_Compiler_Module   : Current module name, if the line is inside a
  module
.
 #PB_Compiler_Version  : Compiler version, in integer format in the
  form '420' for 4.20.
 #PB_Compiler_Home     : Full path of the PureBasic directory, can be
  useful to locate include files

 #PB_Compiler_Debugger : Set to 1 if the runtime debugger
is enabled, set to 0 else. When an executable
                         is created, the debugger is always disabled
  (this constant will be 0).
```

```
 #PB_Compiler_Thread   : Set to 1 if the executable is compiled in
  thread safe mode, set to 0 else.
 #PB_Compiler_Unicode  : Set to 1 if the executable is compiled in
  unicode
mode, set to 0 else.
 #PB_Compiler_LineNumbering : Set to 1 if the executable is compiled
  with OnError line numbering
support, set to 0 else.
 #PB_Compiler_InlineAssembly: Set to 1 if the executable is compiled
  with inline assembly
support, set to 0 else.
 #PB_Compiler_EnableExplicit: Set to 1 if the executable is compiled
  with enable explicit support, set to 0 else.
 #PB_Compiler_IsMainFile    : Set to 1 if the file being compiled is
  the main file, set to 0 else.
 #PB_Compiler_IsIncludeFile : Set to 1 if the file being compiled has
  been included by another file, set to 0 else.
 #PB_Compiler_32Bit    : Set to 1 if the compiler generates 32-bit
  code, set to 0 else.
 #PB_Compiler_64Bit    : Set to 1 if the compiler generates 64-bit
  code, set to 0 else.
 #PB_Compiler_Optimizer: Set to 1 if the compiler generates optimized
  code, set to 0 else.
 #PB_Compiler_DPIAware : Set to 1 if the compiler generates a DPI
  aware executable, set to 0 else.
```

Chapter 24

Compiler Functions

Syntax

```
Size = SizeOf(Type)
```

Description

SizeOf can be used to find the size of any complex Structure , built-in type (word, float, etc.), Interface or even ReferenceLink ”variables” ”variables” (Structures with same name as variable take precedence). This can be useful in many areas such as calculating memory requirements for operations, using API commands, etc.
As a compiler function, SizeOf(x) is assigned to a constant and does not require assignment to another variable if inside a loop or often called procedure.
As a compile time function SizeOf doesn't work with runtime Array, List or Map. ArraySize() , ListSize() or MapSize() can be used instead.
Note: A Character (.c) variable is unicode and uses 2 bytes. An Ascii variable (.a) is Ascii and uses 1 byte.

Parameters

Type The type of the object.

Return value

The size of the object in memory, in bytes.

Example: 1

```
1  char.c='!'
2  Debug SizeOf(char); display 2
3
4  ascii.a='!'
5  Debug SizeOf(ascii); display 1
```

Example: 2

```
Structure Person
  Name.s
  ForName.s
  Age.w
EndStructure

Debug "The size of my friend is "+Str(Sizeof(Person))+" bytes" ; will be 10 on 32-bit compiler as a string pointer is 4 bytes in memory
                                                               ; will be 18 on 64-bit compiler as a string pointer is 8 bytes in memory

John.Person\Name = "John"

Debug SizeOf(John) ; will be the same
```

Syntax

```
Index = OffsetOf(Structure\Field)
Index = OffsetOf(Interface\Function())
```

Description

OffsetOf can be used to find out the address offset of a Structure field or the address offset of an Interface function. When used with an Interface , the function index is the memory offset, so it will be IndexOfTheFunction*SizeOf(Integer).

Parameters

Structure\Field or Interface\Function() The field of the structure or function of the interface.

Return value

Returns the index of the field or function, zero otherwise.

Example

```
Structure Person
  Name.s
  ForName.s
  Age.w
EndStructure

Debug OffsetOf(Person\Age) ; will be  8 on 32-bit compiler as a string pointer is 4 bytes in memory
                           ; will be 16 on 64-bit compiler as a string pointer is 8 bytes in memory

Interface ITest
  Create()
  Destroy(Flags)
EndInterface

```

```
15  Debug OffsetOf(ITest\Destroy()) ; will be 4 on 32-bit compiler as a
     pointer is 4 bytes in memory
16                                  ; will be 8 on 64-bit compiler as a
     pointer is 8 bytes in memory
```

Syntax

```
Type = TypeOf(Object)
```

Description

TypeOf can be used to find out the type of a variable , or a structure field .

Parameters

Object The object to use.

Return value

The type of the object.
The type can be one of the following values:

```
#PB_Byte
#PB_Word
#PB_Long
#PB_String
#PB_Structure
#PB_Float
#PB_Character
#PB_Double
#PB_Quad
#PB_List
#PB_Array
#PB_Integer
#PB_Map
#PB_Ascii
#PB_Unicode
#PB_Interface
```

Example

```
1   Structure Person
2     Name.s
3     ForName.s
4     Age.w
5   EndStructure
6
7   If TypeOf(Person\Age) = #PB_Word
8     Debug "Age is a 'Word'"
9   EndIf
10
11  Surface.f
12  If TypeOf(Surface) = #PB_Float
```

```
13      Debug "Surface is a 'Float'"
14    EndIf
```

Syntax

```
Result = Subsystem(<constant string expression>)
```

Description

Subsystem can be used to find out if a subsystem is in use for the program being compiled. The specified subsystem name is not case-sensitive.

Parameters

constant string expression The subsystem name.
Windows: DirectX9, DirectX11
Linux : Gtk2, Qt
MacOS X: None

Return value

Returns nonzero if the subsystem is used, zero otherwise.

Example

```
1   CompilerIf Subsystem("OpenGL")
2     Debug "Compiling with the OpenGL subsystem"
3   CompilerEndIf
```

Syntax

```
Result = Defined(Name, Type)
```

Description

Defined checks if a particular object of a code source like structure , interface , variables etc. is already defined or not.

Parameters

Name The name of the object. The 'Name' parameter has to be specified without any extra decoration (ie: without the '#' for a constant , without '()' for an array , a list or a map).

Type The 'Type' parameter can be one of the following values:

```
#PB_Constant
#PB_Variable
#PB_Array
#PB_List
#PB_Map
```

```
#PB_Structure
#PB_Interface
#PB_Procedure
#PB_Function
#PB_OSFunction
#PB_Label
#PB_Prototype
#PB_Module
#PB_Enumeration
```

Return value

Returns nonzero if the object is defined, zero otherwise.

Example

```
#PureConstant = 10

CompilerIf Defined(PureConstant, #PB_Constant)
  Debug "Constant 'PureConstant' is declared"
CompilerEndIf

Test = 25

CompilerIf Defined(Test, #PB_Variable)
  Debug "Variable 'Test' is declared"
CompilerEndIf
```

Syntax

```
InitializeStructure(*Pointer, Structure)
```

Description

InitializeStructure initialize the specified structured memory area. It initializes structure members of type Array, List and Map, other members are not affected (.s, .l, .i etc).

Parameters

***Pointer** The memory address to use.

Structure 'Structure' is the name of the structure which should be used to perform the initialization. There is no internal check to ensures the structure match the memory area. Warning: multiple calls to InitializeStructure create a memory leak because the old members are not freed (ClearStructure has to be called before calling InitializeStructure once more). This function is for advanced users and should be used with care. To allocate dynamic structure, use AllocateStructure() ().

Return value

None.

Example

```
Structure People
  Name$
  Age.l
  List Friends.s()
EndStructure

*Student.People = AllocateMemory(SizeOf(People))
InitializeStructure(*Student, People)

; Now the list is ready to use
;
AddElement(*Student\Friends())
*Student\Friends() = "John"

AddElement(*Student\Friends())
*Student\Friends() = "Yann"

; Print out the list content
;
ForEach *Student\Friends()
  Debug *Student\Friends()
Next
```

Syntax

```
Result = CompareStructure(*Pointer1, *Pointer2, Structure [, Flags])
```

Description

CompareStructure compares the memory of two structures for equality. The comparison is recursively applied also to child elements such as arrays , lists , and maps .

Parameters

***Pointer1** The memory address of the first structured variable to be tested.

***Pointer2** The memory address of the second structured variable to be tested.

Structure The structure used.

Flags (optional) It can have one of the following values:

```
#PB_String_CaseSensitive : String comparison is case sensitive
 (a=a). (default)
#PB_String_NoCase        : String comparison is case
 insensitive(a=A).
#PB_Memory_FollowPointers: If a structure element is a pointer
 that is not 0, recursively compare the pointer target.
                           The default is to compare only the
 pointer value itself.
```

Remarks

Caution: The #PB_Memory_FollowPointers option is for advanced users and requires special care to avoid crashes. If this option is used then all pointer values must point to valid and initialized memory or have the value 0. It is also not allowed to have loops in the pointed elements (a chain of pointers that refers back to itself).

Return value

Returns nonzero if the structures are the same or zero if they differ.

Example

```
Structure People
  Name$
  LastName$
  Map Friends$()
  Age.l
EndStructure

StudentA.People\Name$ = "Paul"
StudentA\LastName$ = "Morito"
StudentA\Friends$("Tom") = "Jones"
StudentA\Friends$("Jim") = "Doe"

StudentB.People\Name$ = "Paul"
StudentB\LastName$ = "Morito"
StudentB\Friends$("Tom") = "Jones"
StudentB\Friends$("Jim") = "Doe"

StudentC.People\Name$ = "Paul"
StudentC\LastName$ = "Morito"
StudentC\Friends$("Tom") = "Hanks" ; Not the same as in StudentA
StudentC\Friends$("Jim") = "Doe"

Debug CompareStructure(@StudentA, @StudentB, People) ; Equal
Debug CompareStructure(@StudentA, @StudentC, People) ; Not equal
```

Syntax

```
CopyStructure(*Source, *Destination, Structure)
```

Description

CopyStructure copy the memory of a structured memory area to another. This is useful when dealing with dynamic allocations, through pointers . Every fields will be duplicated, even array , list , and map . The destination structure will be automatically cleared before doing the copy, it's not needed to call ClearStructure before CopyStructure. Warning: the destination should be a valid structure memory area, or a cleared memory area. If the memory area is not cleared, it could crash, as random values will be used by the clear routine. There is no internal check to ensures that the structure match the two memory area. This function is for advanced users and should be used with care.

Parameters

***Source** The memory address containing the structure to copy.

***Destination** The memory address of the copy.

Structure The name of the structure that should be used to perform the copy.

Return value

None.

Example

```
Structure People
  Name$
  LastName$
  Map Friends$()
  Age.l
EndStructure

Student.People\Name$ = "Paul"
Student\LastName$ = "Morito"
Student\Friends$("Tom") = "Jones"
Student\Friends$("Jim") = "Doe"

CopyStructure(@Student, @StudentCopy.People, People)

Debug StudentCopy\Name$
Debug StudentCopy\LastName$
Debug StudentCopy\Friends$("Tom")
Debug StudentCopy\Friends$("Jim")
```

Syntax

```
ClearStructure(*Pointer, Structure)
```

Description

ClearStructure clears a structured memory area. This is useful when the structure contains strings, array, list or map which have been allocated internally by PureBasic. This function is for advanced users and should be used with care.

Parameters

***Pointer** The memory address containing the structure to be erased.

Structure The name of the structure which should be used to perform the clearing. All the fields will be set to zero, even native type like long, integer etc. There is no internal check to ensures the structure match the memory area.

Return value

None.

Example

```
Structure People
  Name$
  LastName$
  Age.l
EndStructure

Student.People\Name$ = "Paul"
Student\LastName$ = "Morito"
Student\Age = 10

ClearStructure(@Student, People)

; Will print empty strings as the whole structure has been cleared.
 All other fields have been reset to zero.
;
Debug Student\Name$
Debug Student\LastName$
Debug Student\Age
```

Syntax

```
ResetStructure(*Pointer, Structure)
```

Description

ResetStructure clears a structured memory area and initialize it to be ready to use. This is useful when the structure contains strings, array, list or map which have been allocated internally by PureBasic. This function is for advanced users and should be used with care.

Parameters

***Pointer** The memory address containing the structure to be reinitialized.

Structure The name of the structure which should be used to perform the clearing.

Return value

None.

Example

```
Structure Person
  Map Friends.s()
EndStructure

```

```
  Henry.Person\Friends("1") = "Paul"

  ResetStructure(@Henry, Person)

  ; Will print an empty string as the whole structure has been reset.
   The map is still usable but empty.
  ;
  Debug Henry\Friends("1")
```

Syntax

```
Result = Bool(<boolean expression>)
```

Description

Bool can be used to evaluate a boolean expression outside of regular conditional operator like If, While, Until etc.

Parameters

boolean expression The Boolean expression to test.

Return value

Returns #True if the boolean expression is true, #False otherwise.

Example

```
  Hello$ = "Hello"
  World$ = "World"

  Debug Bool(Hello$ = "Hello")  ; will print 1
  Debug Bool(Hello$ <> "Hello" Or World$ = "World") ; will print 1
```

Chapter 25

Data

Introduction

PureBasic allows the use of Data, to store predefined blocks of information inside of your program. This is very useful for default values of a program (language string for example) or, in a game, to define the sprite way to follow (precalculated).
DataSection must be called first to indicate a data section follow. This means all labels and data component will be stored in the data section of the program, which has a much faster access than the code section. Data will be used to enter the data. EndDataSection must be specified if some code follows the data declaration. One of good stuff is you can define different Data sections in the code without any problem. Restore and Read command will be used to retrieve the data.
These functions are not thread safe so don't use them inside a thread.

Commands

Syntax

```
DataSection
```

Description

Start a data section.

Example

```
1  DataSection
2    NumericData:
3    Data.w 120, 250, 645 ; 3 'word' sized numbers
4  EndDataSection
```

Syntax

```
EndDataSection
```

Description

End a data section.

Example

```
DataSection
  NumericData:
  Data.w 0, 1, 2
EndDataSection
```

Syntax

```
Data.TypeName
```

Description

Defines data. The type can only be a native basic type (integer, long, word, byte, ascii, unicode, float, double, quad, character, string). Any number of data can be put on the same line, each one delimited with a comma ',’. A literal string cannot exceed 8192 characters.

Example

```
DataSection
  MixedData:
  Data.l 100, 200, -250, -452, 145
  Data.s "Hello", "This", "is ", "What ?"
EndDataSection
```

For advanced programmers: it's also possible to put a procedure address or a label address inside Data when its type is set to integer (.i). (Using the 'integer' type will store the (different) addresses accurate on both 32-bit and 64-bit environments.) This can be used to build easy virtual function tables for example.

Example

```
Procedure Max(Number, Number2)
EndProcedure

Label:

DataSection
  MixedData:
  Data.i ?Label, @Max()
EndDataSection
```

Example

```
Interface MyObject
  DoThis()
  DoThat()
EndInterface

Procedure This(*Self)
  MessageRequester("MyObject", "This")
```

```
  EndProcedure

  Procedure That(*Self)
    MessageRequester("MyObject", "That")
  EndProcedure

  m.MyObject = ?VTable

  m\DoThis()
  m\DoThat()


  DataSection
    VTable:
      Data.i ?Procedures
    Procedures:
      Data.i @This(), @That()
  EndDataSection
```

Syntax

```
Restore label
```

Description

This keyword is useful to set the start indicator for the Read to a specified label. All labels used for this purpose should be placed within the DataSection because the data is treated as a separate block from the program code when it is compiled and may become disassociated from a label if the label were placed outside of the DataSection.

Example

```
  Restore StringData
  Read.s MyFirstData$
  Read.s MySecondData$

  Restore NumericalData
  Read.l a
  Read.l b

  Debug MyFirstData$
  Debug a

  End

  DataSection
    NumericalData:
      Data.l 100, 200, -250, -452, 145

    StringData:
      Data.s "Hello", "This", "is ", "What ?"
  EndDataSection
```

Syntax

```
Read[.<type>] <variable>
```

Description

Read the next available data. The next available data can be changed by using the Restore command. By default, the next available data is the first data declared. The type of data to read is determined by the type suffix. The default type will be used if it is not specified. It is however advisable to use the ad-hoc type in order to avoid the error message that will appear when you read data string and to avoid an integer type confusion which is a 'Long' type for 32-bit compilers and which is a 'Quad' type for 64-bit compilers.

Example

```
Restore StringData
Read.s MyFirstData$

Restore NumericalData
Read a   ; Beware, it will read an Integer (Long in 32-bits or Quad in 64-bits compilers)
Read.q b
Read c   ; Beware, read a 'Quad' in a 64-bit compiler even if the Data is a 'Long'!
Read.l d

Debug MyFirstData$ ; Display Hello
Debug a  ; Display 100
Debug b  ; Display 11111111111111111
Debug c  ; Beware, the display depends of your compiler! : 200 in 32-bits or 1288490189000 in 64-bits
Debug d  ; Idem : 300  in 32-bits or 400 in 64-bits

End

DataSection
  NumericalData:
    Data.i 100
    Data.q 111111111111111111
    Data.l 200, 300, 400

  StringData:
    Data.s "Hello", "This", "is ", "What ?"
EndDataSection
```

Chapter 26

Debugger keywords in PureBasic

Overview

A complete description of all functions of the powerful debugger you will find in the extra chapters Using the debugger or Using the debugging Tools .
Following is a list of special keywords to control the debugger from your source code. There is also a Debugger library which provides further functions to modify the behavior of the debugger should it be present. Several compiler constants useful also for debugging purposes you find in the Compiler directives section.

Syntax

```
CallDebugger
```

Description

This invokes the ”debugger” and freezes the program immediately.

Syntax

```
Debug <expression> [, DebugLevel]
```

Description

Display the DebugOutput window and the result inside it. The expression can be any valid PureBasic expression, from numeric to string. An important point is the Debug command and its associated expression is totally ignored (not compiled) when the debugger is deactivated.
Note: This is also true, if you’re using complete command lines after Debug (e.g. Debug LoadImage(1,”test.bmp”)). They will not be compiled with disabled debugger!
This means this command can be used to trace easily in the program without having to comment all the debug commands when creating the final executable.
The ’DebugLevel’ is the level of priority of the debug message. All normal debug message (without specifying a debug level) are automatically displayed. When a level is specified then the message will be only displayed if the current DebugLevel (set with the following DebugLevel command) is equals or above this number. This allows hierarchical debug mode, by displaying more and more precise information in function of the used DebugLevel.

Syntax

```
DebugLevel <constant expression>
```

Description

Set the current debug level for the 'Debug' message.
Note: The debug level is set at compile time, which means you have to put the DebugLevel command before any other Debug commands to be sure it affects them all.

Syntax

```
DisableDebugger
```

Description

This will disable the debugger checks on the source code which follow this command. It doesn't not fully turn off the debugger, so performance checks should not be done using DisableDebugger command, but by disabling the debugger before compiling the program.

Syntax

```
EnableDebugger
```

Description

This will enable the debugger checks on the source code which follow this command (if the debugger was previously disabled with DisableDebugger).
Note: EnableDebugger doesn't have an effect, if the debugger is completely disabled in the IDE (look at Compiler settings).

Chapter 27

Define

Syntax

```
Define.<type> [<variable> [= <expression>], <variable> [=
    <expression>], ...]
```

Description

Assign the same data type to a series of variables .
Without this keyword, variables are created with the default type of PureBasic which is the type INTEGER. As a reminder, the type INTEGER is:
4 bytes (with a 32-bit compiler) ranging from -2147483648 to + 2147483647
8 bytes (with a 64-bit compiler) ranging from -9223372036854775808 to +9223372036854775807
Define is very flexible because it also allows you to assign a different type to a particular variable within a series.
Define can also be used with arrays , the lists and the maps .

Example: Serial assignment

```
1   Define.b a, b = 10, c = b*2, d ; these 4 variables will be byte
    typed (.b)
```

Example: Mix of individual and serial assignments

```
1   Define.q a, b.w, c, d, st.s = "ok"  ; a, c, and d are Quad (.q), b
    is a Word (.w) and st a String (.s)
2
3   Debug SizeOf(a)  ; will print 8
4   Debug SizeOf(b)  ; will print 2, because it doesn't have the default
    type
5   Debug SizeOf(c)  ; will print 8
6   Debug SizeOf(d)  ; will print 8
7   Debug st         ; will print ok
```

Syntax

```
Define <variable>.<type> [= <expression>] [, <variable>.<type> [=
    <expression>], ...]
```

Description

Alternative possibility for the variable declaration using Define.

Example

```
1  Define MyChar.c
2  Define MyLong.l
3  Define MyWord.w
4
5  Debug SizeOf(MyChar)    ; will print 2
6  Debug SizeOf(MyLong)    ; will print 4
7  Debug SizeOf(MyWord)    ; will print 2
```

Chapter 28

Dim

Syntax

```
Dim name.<type>(<expression>, [<expression>], ...)
```

Description

Dim is used to create new arrays (the initial value of each element will be zero). An array in PureBasic can be of any types, including structured , and user defined types. Once an array is defined it can be resized with ReDim. Arrays are dynamically allocated which means a variable or an expression can be used to size them. To view all commands used to manage arrays, see the Array library.
When you define a new array, please note that it will have one more element than you used as parameter, because the numbering of the elements in PureBasic (like in other BASIC's) starts at element 0. For example when you define Dim(10) the array will have 11 elements, elements 0 to 10. This behavior is different for static arrays in structures . Static arrays use brackets "[]", for example ArrayStatic[2] has only 2 elements from 0 to 1 and library functions Array don't work with them.
The new arrays are always locals, which means Global or Shared commands have to be used if an array declared in the main source need to be used in procedures. It is also possible to pass an array as parameter to a procedure - by using the keyword Array. It will be passed "by reference" (which means, that the array will not be copied, instead the functions in the procedure will manipulate the original array).
To delete the content of an array and release its used memory during program flow, call FreeArray() .
If Dim is used on an existing array, it will reset its contents to zero.
For fast swapping of array contents the Swap keyword is available.
Note: Array bound checking is only done when the runtime Debugger is enabled.

Example

```
1  Dim MyArray(41)
2  MyArray(0) = 1
3  MyArray(1) = 2
```

Example: Multidimensional array

```
1  Dim MultiArray.b(NbColumns, NbLines)
2  MultiArray(10, 20) = 10
3  MultiArray(20, 30) = 20
```

Example: Array as procedure parameter

```
Procedure fill(Array Numbers(1), Length)  ; the 1 stands for the number of dimensions in the array
  For i = 0 To Length
    Numbers(i) = i
  Next
EndProcedure

Dim Numbers(10)
fill(Numbers(), 10)  ; the array Numbers() will be passed as parameter here

Debug Numbers(5)
Debug Numbers(10)
```

Syntax

```
ReDim name.<type>(<expression>, [<expression>], ...)
```

Description

ReDim is used to 'resize' an already declared array while preserving its content. The new size can be larger or smaller, but the number of dimension of the array cannot be changed.
If ReDim is used with a multi-dimension array, only its last dimension can be changed.

Example

```
Dim MyArray.l(1) ; We have 2 elements
MyArray(0) = 1
MyArray(1) = 2

ReDim MyArray(4) ; Now we want 5 elements
MyArray(2) = 3

For k = 0 To 2
  Debug MyArray(k)
Next
```

Example: Multi-dimension array

```
Dim MyTab.l(1,1,1)

;  ReDim MyTab(4,1,1) ; NO !
;  ReDim MyTab(1,4,1) ; NO !
ReDim MyTab(1,1,4) ; YES, only its last dimension can be changed!
MyTab(1,1,4) = 3
```

Chapter 29

Building a DLL

PureBasic allows to create standard Microsoft Windows DLL (Dynamic Linked Library), shared objects (.so) on Linux, and dynamic libraries (.dylib) on MacOS X. The DLL code is like a PureBasic code excepts than no real code should be written outside of procedure.
When writing a DLL, all the code is done inside procedures. When a procedure should be public (ie: accessible by third programs which will use the DLL), the keyword ProcedureDLL (or ProcedureCDLL if the procedure needs to be in 'CDecl' format, which is not the case of regular Windows DLL) is used instead of Procedure (and DeclareDLL or DeclareCDLL if are used instead of Declare). This is the only change to do to a program.
When this is done, select 'Shared DLL' as output format ('Compiler Option' window in the PureBasic editor or /DLL switch in command line) and a DLL with the name you set (in the save-requester when using the IDE) will be created in the selected directory.

Example

```
1   ProcedureDLL MyFunction()
2     MessageRequester("Hello", "This is a PureBasic DLL !", 0)
3   EndProcedure
4
5   ; Now the client program, which use the DLL
6   ;
7   If OpenLibrary(0, "PureBasic.dll")
8     CallFunction(0, "MyFunction")
9     CloseLibrary(0)
10  EndIf
```

For advanced programmers: there is 4 special procedures which are called automatically by the OS when one of the following events happen:
- DLL is attached to a new process
- DLL is detached from a process
- DLL is attached to a new thread
- DLL is detached from a thread

To handle that, it's possible to declare 4 special procedures called: AttachProcess(Instance), DetachProcess(Instance), AttachThread(Instance) and DetachThread(Instance). This means these 4 procedures names are reserved and can't be used by the programmer for other purposes.
Notes about creating DLL's:
- The declaration of arrays, lists or map with Dim , NewList or NewMap must always be done inside the procedure AttachProcess.
- Don't write program code outside procedures. The only exception is the declaration of variables or structures.
- Default values in procedure parameters have no effect.

- DirectX initialization routines must not be written in the AttachProcess procedure.

Note about returning strings from DLL's:

If you want to return a string out of a DLL, the string has to be declared as Global before using it.

Example

```
Global ReturnString$

ProcedureDLL.s MyFunction(var.s)
  ReturnString$ = var + " test"
  ProcedureReturn ReturnString$
EndProcedure
```

Without declaring it as Global first, the string is local to the ProcedureDLL and can't be accessed from outside.

When using CallFunction() (or one of its similar CallXXX functions) on a DLL function you will get a pointer on the return string, which you could read with PeekS() .

Example

```
String.s = PeekS(CallFunction(0, "FunctionName", Parameter1,
 Parameter2))
```

Here a complete code example:

Chapter 30

Enumerations

Syntax

```
Enumeration [name] [<constant> [Step <constant>]]
  #Constant1
  #Constant2 [= <constant>]
  #Constant3
  ...
EndEnumeration



EnumerationBinary [name] [<constant>]
  #Constant1
  #Constant2 [= <constant>]
  #Constant3
  ...
EndEnumeration
```

Description

Enumerations are very handy to declare a sequence of constants quickly without using fixed numbers. The first constant found in the enumeration will get the number 0 and the next one will be 1 etc. It's possible to change the first constant number and adjust the step for each new constant found in the enumeration. If needed, the current constant number can be altered by affecting with '=' the new number to the specified constant. As Enumerations only accept integer numbers, floats will be rounded to the nearest integer.
A name can be set to identify an enumeration and allow to continue it later. It is useful to group objects altogether while declaring them in different code place.
For advanced user only: the reserved constant #PB_Compiler_EnumerationValue store the next value which will be used in the enumeration. It can be useful to get the last enumeration value or to chain two enumerations.
EnumerationBinary can be used to create enumerations suitable for flags. The first item value is 1.

Example: Simple enumeration

```
1  Enumeration
2    #GadgetInfo ; will be 0
3    #GadgetText ; will be 1
4    #GadgetOK   ; will be 2
5  EndEnumeration
```

Example: Enumeration with step

```
Enumeration 20 Step 3
  #GadgetInfo ; will be 20
  #GadgetText ; will be 23
  #GadgetOK   ; will be 26
EndEnumeration
```

Example: Enumeration with dynamic change

```
Enumeration
  #GadgetInfo       ; will be 0
  #GadgetText = 15  ; will be 15
  #GadgetOK         ; will be 16
EndEnumeration
```

Example: Named enumeration

```
Enumeration Gadget
  #GadgetInfo ; will be 0
  #GadgetText ; will be 1
  #GadgetOK   ; will be 2
EndEnumeration

Enumeration Window
  #FirstWindow  ; will be 0
  #SecondWindow ; will be 1
EndEnumeration

Enumeration Gadget
  #GadgetCancel ; will be 3
  #GadgetImage  ; will be 4
  #GadgetSound  ; will be 5
EndEnumeration
```

Example: Getting next enumeration value

```
Enumeration
  #GadgetInfo ; will be 0
  #GadgetText ; will be 1
  #GadgetOK   ; will be 2
EndEnumeration

Debug "Next enumeration value: " + #PB_Compiler_EnumerationValue ;
  will print 3
```

Example: Binary enumeration

```
EnumerationBinary
  #Flags1 ; will be 1
  #Flags2 ; will be 2
```

```
    #Flags3 ; will be 4
    #Flags4 ; will be 8
    #Flags5 ; will be 16
  EndEnumeration
```

Chapter 31

For : Next

Syntax

```
For <variable> = <expression1> To <expression2> [Step <constant>]
  ...
Next [<variable>]
```

Description

For : Next is used to create a loop within a program with the given parameters. At each loop the <variable> value is increased by a 1, (or of the "Step value" if a Step value is specified) and when the <variable> value is above the <expression2> value, the loop stop.
With the Break command its possible to exit the For : Next loop at any moment, with the Continue command the end of the current iteration can be skipped.
The For : Next loop works only with integer values, at the expressions as well at the Step constant. The Step constant can also be negative.

Example

```
1  For k = 0 To 10
2    Debug k
3  Next
```

In this example, the program will loop 11, time (0 to 10), then quit.

Example

```
1  For k = 10 To 1 Step -1
2    Debug k
3  Next
```

In this example, the program will loop 10 times (10 to 1 backwards), then quit.

Example

```
1  a = 2
2  b = 3
```

```
3   For k = a+2 To b+7 Step 2
4     Debug k
5   Next k
```

Here, the program will loop 4 times before quitting, (k is increased by a value of 2 at each loop, so the k value is: 4-6-8-10). The "k" after the "Next" indicates that "Next" is ending the "For k" loop. If another variable, is used the compiler will generate an error. It can be useful to nest several "For/Next" loops.

Example

```
1   For x=0 To 10
2     For y=0 To 5
3       Debug "x: " + x + " y: " + y
4     Next y
5   Next x
```

Note: Be aware, that in PureBasic the value of <expression2> ('To' value) can also be changed inside the For : Next loop. This can lead to endless loops when wrongly used.

Chapter 32

ForEach : Next

Syntax

```
ForEach List() Or Map()
  ...
Next [List() Or Map()]
```

Description

ForEach loops through all elements in the specified list or map starting from the first element up to the last. If the list or the map is empty, ForEach : Next exits immediately. To view all commands used to manage lists, please click here . To view all commands used to manage maps, please click here .
When used with list, it's possible to delete or add elements during the loop. As well it's allowed to change the current element with ChangeCurrentElement() . After one of the mentioned changes the next loop continues with the element following the current element.
When used with map, please do not put a key (see example below).
With the Break command its possible to exit the ForEach : Next loop at any moment, with the Continue command the end of the current iteration can be skipped.

Example: list

```
1  NewList Number()
2
3  AddElement(Number())
4  Number() = 10
5
6  AddElement(Number())
7  Number() = 20
8
9  AddElement(Number())
10 Number() = 30
11
12 ForEach Number()
13   Debug Number() ; Will output 10, 20 and 30
14 Next
```

Example: Map

```
1   NewMap Country.s()
2
3   Country("US") = "United States"
4   Country("FR") = "France"
5   Country("GE") = "Germany"
6
7   ForEach Country()
8     Debug Country()
9    ;Debug Country("FR") ; No, do not put a key because sometimes it can
      cause an infinite loop
10  Next
```

Chapter 33

General Syntax Rules

PureBasic has established rules which never change. These are:

Comments

Example

```
1   If a = 10 ; This is a comment to indicate something.
```

Keywords

All **keywords** are used for general things inside PureBasic, like creating arrays (Dim) or lists (NewList), or controlling the program flow (If : Else : EndIf). They are not followed by the brackets '()', which are typically used for PureBasic **functions**.

Example

```
1     If a = 1       ; If, Else and EndIf are keywords; while 'a = 1'
2       ...          ; is a variable used inside an expression.
3     Else
4       ...
5     EndIf
```

Keywords are regularly described in the chapters on the left side of the index page in the reference manual.

Functions

Every **function** must be followed by an opening round bracket '(' character, otherwise it will not be considered a function (this applies also to functions without parameters).

Example

```
1    EventWindow() ; it is a function.
2    EventWindow   ; it is a variable.
```

Functions are regularly included in the PureBasic "Command libraries", described on the right side of the index page in the reference manual.

Constants

All constants are preceded by the hash # character. They can only be declared once in the source and always keep their predefined values. (The compiler replaces all constant names with their corresponding values when compiling the executable.)

Example

```
1  #Hello = 10 ; it is a constant.
2  Hello  = 10 ; it is a variable.
```

Literal strings

Literal strings are declared using the " character. Escape sequences are supported by prepending the * * character to the literal string. The supported escape sequences are:

```
\a: alarm           Chr(7)
\b: backspace       Chr(8)
\f: formfeed        Chr(12)
\n: newline         Chr(10)
\r: carriage return Chr(13)
\t: horizontal tab  Chr(9)
\v: vertical tab    Chr(11)
\": double quote    Chr(34)
\\: backslash       Chr(92)
```

There are two special constants for strings:

```
#Empty$: represents an empty string (exactly the same as "")
#Null$ : represents an null string. This can be used for API
         functions requiring a null pointer to a string, or to really
 free a string.
```

Warning: On Windows, \t does not work with the graphical functions of the 2DDrawing and VectorDrawing libraries.

Example

```
1  a$ =  "Hello world"  ; standard string
2  b$ = ~"Escape\nMe !" ; string with escape sequences
```

Labels

All labels must be followed by a colon : character. Label names may not contain any operators (+,-,...) or special characters (ß,ä,ö,ü,...). Labels defined inside a procedure will be available only in that procedure.

Example

```
I_am_a_label:
```

Modules

Modules use a :: (double colon) when referencing a module name. Be careful not to confuse these with the single colon used by labels.

Example

```
Debug CarModule::NbCars
```

Expressions

An expression is something which can be evaluated. An expression can mix any variables, constants, or functions, of the same type. When using numbers in an expression, you can prefix them with the $ symbol to indicate a hexadecimal number, or with the % symbol to indicate a binary number. Without either of those, the number will be treated as decimal. Strings must be enclosed within straight double quotes.

Example

```
a = a + 1 + (12 * 3)

a = a + WindowHeight(#Window) + b/2 + #MyConstant

If a <> 12 + 2
  b + 2 >= c + 3
EndIf

a$ = b$ + "this is a string value" + c$

Foo = Foo + $69 / %1001  ; Hexadecimal and binary number usage
```

Concatenation of commands

Any number of commands can be added to the same line by using the : option.

Example

```
If IsCrazy = 0 : MessageRequester("Info", "Not Crazy") : Else :
 MessageRequester("Info", "Crazy") : EndIf
```

Line continuation

Long expressions can be split across several lines. A split line has to end with one of the following operators: plus (+), comma (,), or (||), And, Or, Xor.

Example

```
Text$ = "Very very very very long text" + #LF$ +
        "another long text" + #LF$ +
        " and the end of the long text"

MessageRequester("Hello this is a very long title",
                 "And a very long message, so we can use the
 multiline" + #LF$ + Text$,
                 #PB_MessageRequester_Ok)
```

Typical words in this manual

Words used in this manual:

<variable> : a basic variable.
<expression> : an expression as explained above.
<constant> : a numeric constant.
<label> : a program label.
<type> : any type, (standard or structured).

Others

- In this guide, all topics are listed in alphabetical order to decrease any search time.
- **Return values** of commands are always Integer if no other type is specified in the Syntax line of the command description.
- In the PureBasic documentation, the terms ”commands” and ”functions” are used interchangeably, regardless of whether the function returns a value or not. To learn whether value is returned by a specific command or function, consult the description provided in the command’s documentation.

Chapter 34

Global

Syntax

```
Global[.<type>] <variable[.<type>]> [= <expression>] [, ...]
```

Description

Global provides the ability for variables to be defined as global, i.e., variables defined as such may then be accessed within a Procedure . In this case the command Global must be called for the according variables, **before** the declaration of the procedure. This rule is true everywhere except in a single case: The modules do not have access to the global declared variables outside this module. Each variable may have a default value directly assigned to it. If a type is specified for a variable after Global, the default type is changed through the use of this declaration. Global may also be used with arrays , lists and maps .
In order to use local variables within a procedure, which have the same name as global variables, take a look at the Protected and Static keywords.

Example: With variables

```
1  Global a.l, b.b, c, d = 20
2
3  Procedure Change()
4    Debug a  ; Will be 10 as the variable is global
5  EndProcedure
6
7  a = 10
8  Change()
```

Example: With array

```
1  Global Dim Array(2)
2
3  Procedure Change()
4    Debug Array(0)  ; Will be 10 as the array is global
5  EndProcedure
6
7  Array(0) = 10
8  Change()
```

Example: With default type

```
; 'Angle' and 'Position' will be a float, as they didn't have a specified type
;
Global.f Angle, Length.b, Position
```

Example: Complex with a module

```
Global Var_GlobalOutsideModule = 12
Debug  Var_GlobalOutsideModule ; Display 12

DeclareModule Ferrari
  Global Var_GlobalModule = 308
  #FerrariName$ = "458 Italia"
  Debug #FerrariName$ ; Display "458 Italia"

  ; Debug Var_GlobalOutsideModule ; Error, the variable already exists
  Debug Var_GlobalModule       ; Display 308

  Declare CreateFerrari()
EndDeclareModule


; Private
; ------------------------------------------------------------------------------
Module Ferrari
  Debug Var_GlobalOutsideModule  ; Display 0 <== Look !
  Debug Var_GlobalModule      ; Display 308

  Global Initialized = 205
  Debug Initialized ; Display 205

  Procedure Init()
    Debug Var_GlobalOutsideModule ; Display 0
    Debug Var_GlobalModule     ; Display 308
    Debug "InitFerrari()"
  EndProcedure

  Procedure CreateFerrari()    ; Public
    Init()
    Debug "CreateFerrari()"
    Debug Var_GlobalOutsideModule ; Display 0
    Debug Var_GlobalModule     ; Display 308
  EndProcedure

EndModule


; Main Code
; ------------------------------------------------------------------------------
Procedure Init()

  Debug " init() from main code."
  Debug Var_GlobalOutsideModule ; Display 12
```

```
47      Debug Var_GlobalModule      ; Display 0
48    EndProcedure
49
50    Init()
51
52    Ferrari::CreateFerrari()
53    Debug Ferrari::#FerrariName$ ; Display 458 Italia
54    Debug Var_GlobalOutsideModule  ; Display 12
55    ; Debug Var_GlobalModule     ;  Error, the variable already exists
56
57
58    UseModule Ferrari
59
60    CreateFerrari()
61    Debug #FerrariName$          ; Display 458 Italia
62    Debug Var_GlobalOutsideModule ; Display 12
63    Debug Var_GlobalModule      ; Display 308 <== Look !
64
65    UnuseModule Ferrari
66    ; Debug #FerrariName$        ; Error, does not exist
67    Debug Var_GlobalOutsideModule ; Display 12
68    Debug Var_GlobalModule      ; Display 0 <== Look !
```

Chapter 35

Gosub : Return

Syntax

```
Gosub MyLabel

MyLabel:
  ...
Return
```

Description

Gosub stands for 'Go to sub routine'. A label must be specified after Gosub, at that point the program execution continues immediately after the position defined by that label, and will do so until encountering a Return. When a return is reached, the program execution is then transferred immediately below the Gosub.
Gosub is useful when building fast structured code.
Another technique which may be used in order to insert a sub routine into a standalone program component is to use procedures . Gosub may only be used within the main body of the source code, and may not be used within procedures .

Example

```
1   a = 1
2   b = 2
3   Gosub ComplexOperation
4   Debug a
5   End
6
7   ComplexOperation:
8     a = b*2+a*3+(a+b)
9     a = a+a*a
10  Return
```

Syntax

```
FakeReturn
```

Description

If the command Goto is used within the body of a sub routine, FakeReturn must be used. FakeReturn simulates a return without actually executing a return, and if it is not used, the program will crash. Note: To exit a loop safely, Break should be used instead of Goto.

Example

```
1  Gosub SubRoutine1
2
3  SubRoutine1:
4    ...
5    If a = 10
6      FakeReturn
7      Goto Main_Loop
8    EndIf
9  Return
```

Chapter 36

Handles and Numbers

Numbers

All created objects are identified by an arbitrary number (which is not the object's handle, as seen below). In this manual, these numbers are marked as #Number (for example, each gadget created have a #Gadget number).
The numbers you assign to them do not need to be constants, but they need to be unique for each object in your program (an image can get the same number as a gadget, because these are different types of objects). These numbers are used to later access these objects in your program.
For example, the event handling functions return these numbers:

```
1  EventGadget()
2  EventMenu()
3  EventWindow()
```

Handles

All objects also get a unique number assigned to them by the system. These identifiers are called handles. Sometimes, a PureBasic function doesn't need the number as argument, but the handle. In this manual, such things are mentioned, as an ID.

Example

```
1  ImageGadget(#Gadget, x, y, Width, Height, ImageID [, Flags])
2  ; #Gadget needs to be the number you want to assign to the Gadget
3  ; ImageID needs to a handle to the image.
```

To get the handle to an object, there are special functions like:

```
1  FontID()
2  GadgetID()
3  ImageID()
4  ThreadID()
5  WindowID()
```

Also, most of the functions that create these objects also return this handle as a result, if they were successful. This is only the case if #PB_Any was not used to create the object. If #PB_Any is used, these commands return the object number that was assigned by PB for them, not the handle.

Example

```
GadgetHandle = ImageGadget(...)
```

Chapter 37

If : Else : EndIf

Syntax

```
If <expression>
  ...
[ElseIf <expression>]
  ...
[Else]
  ...
EndIf
```

Description

The If structure is used to achieve tests, and/or change the programmes direction, depending on whether the test is true or false. ElseIf optional command is used for any number of additional tests if the previous test was not true. The Else optional command is used to execute a part of code, if all previous tests were false. Any number of If structures may be nested together.
Short-circuit evaluations for expressions are supported, meaning if a test is true, all following tests will be ignored and not even run.

Example: Basic test

```
1  a = 5
2  If a = 10
3    Debug "a = 10"
4  Else
5    Debug "a <> 10"
6  EndIf
```

Example: Multiple test

```
1  a = 10
2  ; b = 15
3  c = 20
4
5  If (a = 10) And (b >= 10) Or (c = 20)
6    If b = 15
7      Debug "b = 15"
8    Else
```

```
      Debug "Other possibility"
    EndIf
  Else
    Debug "Test failure"
  EndIf
```

Example: Short-circuit test

```
Procedure DisplayHello()
  Debug "Hello"
  ProcedureReturn 1
EndProcedure

a = 10
If a = 10 Or DisplayHello() = 1 ; a is equal to 10, so the second test is fully ignored
  Debug "Test success"
Else
  Debug "Test failure"
EndIf
```

Chapter 38

Import : EndImport

Syntax

```
Import "Filename"
  FunctionName.<type>(<parameter>, [, <parameter> [= DefaultValue]...])
   [As "SymbolName"]
  ...
  VariableName.<type> [As "SymbolName"]
EndImport
```

Description

For advanced programmers. Import : EndImport allows to easy declare external functions and variables from a library (.lib) or an object (.obj) file.
Once declared, the imported functions are directly available for use in the program, like any other commands. The compiler doesn't check if the functions really exists in the imported file, so if an error occurs, it will be reported by the linker.
This feature can replace the OpenLibrary() /CallFunction() sequence as it has some advantages: type checking is done, number of parameters is validated. Unlike CallFunction(), it can deal with double, float and quad without any problem.
The last parameters can have a default value (need to be a constant expression), so if these parameters are omitted when the function is called, the default value will be used.
By default the imported function symbol is 'decorated' in the following way: _FunctionName@callsize. That should work for most of the functions which use the standard call convention (stdcall). If the library is a C one, and the function are not stdcall, the ImportC variant should be used instead. In this case, the default function symbol is decorated like: _FunctionName.
The pseudotypes can be used for the parameters, but not for the returned value.

Remarks

On x64, there is only one calling convention, so ImportC will behave the sames as Import.

Example

```
1  CompilerIf #PB_Compiler_OS <> #PB_OS_Windows
2    CompilerError "This sample only works on Windows"
3  CompilerEndIf
4
5  Import "User32.lib"
```

```
6
7       ; No need to use 'As' as PureBasic decorates the function correctly
8       ; We also define the 'Flags' as optional, with a default value of 0
        (when omitted)
9       ;
10      MessageBoxW(Window.i, Body$, Title$, Flags.i = 0)
11
12    EndImport
13
14    MessageBoxW(0, "Hello", "World") ; We don't specify the flags
```

Example: With pseudotypes

```
1     CompilerIf #PB_Compiler_OS <> #PB_OS_Windows
2       CompilerError "This sample only works on Windows"
3     CompilerEndIf
4
5     Import "User32.lib"
6
7       ; We use the 'p-unicode' pseudotype for the string parameters, as
8       ; MessageBoxW() is an unicode only function. The compiler will
9       ; automatically converts the strings to unicode when needed.
10      ;
11      MessageBoxA(Window.i, Body.p-ascii, Title.p-ascii, Flags.i = 0)
12
13    EndImport
14
15    MessageBoxA(0, "Hello", "World")
```

Chapter 39

Includes Functions

Syntax

```
IncludeFile  "Filename"
```

Description

IncludeFile will always include the specified source file, at the current place in the code (even if XIncludeFile has been called for this file before).

Example

```
1  IncludeFile "Sources\myfile.pb" ; This file will be inserted in the
     current code.
```

This command is useful, if you want to split your source code into several files, to be able to reuse parts e.g. in different projects.

Syntax

```
XIncludeFile "Filename"
```

Description

XIncludeFile is similar to IncludeFile excepts it avoids to include the same file twice.

Example

```
1  XIncludeFile "Sources\myfile.pb" ; This file will be inserted.
2  XIncludeFile "Sources\myfile.pb" ; This file will be ignored along
     with all subsequent calls.
```

This command is useful, if you want to split your source code into several files, to be able to reuse parts e.g. in different projects.

Syntax

```
IncludeBinary "filename"
```

Description

IncludeBinary will include the named file at the current place in the code. Including should be done inside a Data block.

Example

```
1  DataSection
2    MapLabel:
3    IncludeBinary "Data\map.data"
4  EndDataSection
```

This command is especially useful in combination with the Catch-commands (currently there are CatchImage() , CatchSound() , CatchSprite()) to include images, sounds, sprites etc. into the executable.

Syntax

```
IncludePath "path"
```

Description

IncludePath will specify a default path for all files included after the call of this command. This can be very handy when you include many files which are in the same directory.

Example

```
1  IncludePath  "Sources\Data"
2  IncludeFile  "Sprite.pb"
3  XIncludeFile "Music.pb"
```

Chapter 40

Inline x86 ASM

Introduction

Using inline assembly code is not done the same way with the ASM backend and the C backend. With the compiler using the ASM backend, PureBasic allows you to include any x86 assembler commands (including MMX and FPU one) directly in the source code, as if it was a real assembler (X86 and X64 processors only). And it gives you even more: you can use directly any variables or pointers in the assembler keywords, you can put any ASM commands on the same line, ... On Windows and Linux, PureBasic uses **fasm** (http://flatassembler.net), so if you want more information about the syntax, just read the **fasm** guide.
On OS X, PureBasic uses **yasm** (http://yasm.tortall.net/), so if you want more information about the syntax, just read the **yasm** guide.
To activate the inline assembler use the compiler directives EnableASM and DisableASM .
It's possible to enable the ASM syntax coloring in the IDE with the "enable inline ASM syntax coloring" compiler option .

Rules

You have several rules to closely follow if you want to include ASM in a BASIC code :
- The used variables or pointers must be declared **before** to use them in an assembler keyword. Their names in assembler are 'v_variablename' and 'p_pointername', and in a procedure their names are 'p.v_variablename' and 'p.p_pointername'.
- Labels: The labels need to be referenced in lowercase when using the in-line ASM. When you reference a label , you must put the prefix 'l_' before the name.
If the label is defined in a procedure , then its prefix is 'll_procedurename_', in lowercase.
When you reference a module item, you must put the prefix 'module_name.l_' all in lowercase before the item.
If the label is defined in a procedure inside a module, then its prefix is 'module_name.ll_procedurename_', in lowercase.

Example

```
1  DeclareModule MyModule
2    LabelDeclareModule: ;Its name is mymodule.l_labeldeclaremodule:
3    Declare Init()
4  EndDeclareModule
5
6  Module MyModule
7    Procedure Init()
```

```
8       LabelModuleProcedure: ; Its name is
        mymodule.ll_init_labelmoduleprocedure:
9       Debug "InitFerrari()"
10    EndProcedure
11
12    LabelModule1: ;Its name is mymodule.l_labelmodule1:
13  EndModule
14
15  Procedure Test (*Pointer, Variable)
16    TokiSTART:  ;Its name is ll_test_tokistart:
17
18    ! MOV dword [p.p_Pointer], 20
19    ! MOV dword [p.v_Variable], 30
20    Debug *Pointer  ;Its name is p.p_Pointer
21    Debug Variable  ;Its name is p.v_Variable
22  EndProcedure
23
24  VAR=1                          ;Its name is v_VAR
25  *Pointt=AllocateMemory(10)  ;Its name is p_Pointt
26
27  MyModule::Init()
28  Test(0,0)
29
30  Label1: ;Its name is l_label1:
31
32  !jmp l_labelend ; An instruction in assembler has to use the rules
        above. Here it's l_namelabel
33  ;...
34  LabelEnd: ;Its name is l_labelend:
```

- The errors in an ASM part are not reported by PureBasic but by FASM. Just check your code if a such error happen.
- With enabled InlineASM you can't use ASM keywords as label names in your source.
- On x86 processors, the available volatile registers are: eax, ecx and edx, xmm0, xmm1, xmm2 and xmm3. All others must be always preserved.
- On x64 processors, the available volatile registers are: rax, rcx, rdx, r8, r9, xmm0, xmm1, xmm2 and xmm3. All others must be always preserved.
- Windows only: an ASM help-file could be downloaded here. If you place the 'ASM.HLP' in the 'Help/' folder of PureBasic, you can also get help on ASM keywords with F1. (Note: this feature is only enabled, when InlineASM is enabled).

When using assembler in a procedure , you have to be aware of several important things:
- To return directly the 'eax' (or 'rax' on x64) register content, just use ProcedureReturn, without any expression. It will let the 'eax' (or 'rax' on x64) register content untouched and use it as return-value.

Example

```
1   Procedure.l MyTest()
2     MOV eax, 45
3     ProcedureReturn  ; The returned value will be 45
4   EndProcedure
```

- Local variables in PureBasic are directly indexed by the stack pointer, which means if the stack pointer change via an ASM instruction (like PUSH, POP etc..) the variable index will be wrong and direct variable reference won't work anymore.
- It's possible to pass directly an assembly line to the assembler without being processed by the compiler by using the '!' character at the line start. This allow to have a full access to the assembler directives.

When using this, it's possible to reference the local variables using the notation 'p.v_variablename' for a regular variable or 'p.p_variablename' for a pointer.

Example

```
  Procedure Test(*Pointer, Variable)
    ! MOV dword [p.p_Pointer], 20
    ! MOV dword [p.v_Variable], 30
    Debug *Pointer
    Debug Variable
  EndProcedure

  Test(0, 0)
```

"Features of inline assembler with compiler using C backend"

You can use the gcc syntax: https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html.

Example: Pi

```
pi   Procedure.d get_pi()

    Define.d pi

    !asm (
    !    "fldpi\n"
    !    "fstpl   %[v_pi]\n"
    !    :[v_pi]"=m"(v_pi)::
    !);

    ProcedureReturn pi
  EndProcedure

  Define.d n
  n=get_pi()
  Debug n
```

Example: bswap

```
  Procedure bswap(v.l)
    Protected ret.l
    CompilerIf #PB_Compiler_Backend = #PB_Backend_C
      !".intel_syntax noprefix";
      !"mov eax, v_v";
      !"bswap eax";
      !"mov v_ret, eax";
    CompilerElse
      !mov eax, [p.v_v]
      !bswap eax
      !mov [p.v_ret],eax
    CompilerEndIf
    ProcedureReturn ret
```

```
14    EndProcedure
15
16    x.i = $FF000000
17    Debug RSet(Hex(x),8,"0")
18    x = bswap(x)
19    Debug RSet(Hex(x),8,"0")
```

Example: Fibonacci

Procedure.q fib(ub.l, *f) If (ub<3) ProcedureReturn 1 EndIf ; !set_dpfpu(); !asm (! ”mov %[p_f], %%rax;” ! ”fldz;” ! ”fistl (%%rax);” ! ”add $8, %%rax;” ! ”fld1;” ! ”fistl (%%rax);” ! ”add $8, %%rax;” ! ”mov %[v_ub], %%ecx;” ! ”sub $2, %%ecx;” ! ”fib2:” ! ”fxch %%st(1);” ! ”fadd %%st(1),%%st(0);” ! ”fld %%st(0);” ! ”fistpq (%%rax);” ! ”add $8, %%rax;” ! ”dec %%ecx;” ! ”jg fib2;” ! ”fstp %%st(0);” ! ”fstp %%st(0);” ! :[p_f]”=m”(p_f) ! :[v_ub]”m”(v_ub) ! :”rax”,”ecx” !); ProcedureReturn EndProcedure Dim fibonacci.q(20) n=20 fib(n+1, @fibonacci(0)) For i.l=0 To n Debug fibonacci(i) Next

Chapter 41

Interfaces

Syntax

```
Interface <name> [Extends <name>]
  <Method[.<type>]()>
  ...
EndInterface
```

Description

Interfaces are used to access Object Oriented modules, such as COM (Component Object Model) or DirectX dynamic libraries (DLL). These types of libraries are becoming more and more common in Windows, and through the use of interfaces, the ability to access these modules easily (and without any performance hit) is realized. It also introduces the necessary basis for Object Oriented programming within PureBasic, but the use of interfaces requires some advanced knowledge. Most of the standard Windows interfaces have already been implemented within a resident file and this allows direct use of these objects.
The optional Extends parameter may be used to extend another interface with new functions (these functions are commonly called 'methods' in Object Oriented (OO) languages such as C++ or Java). All functions contained within the extended interface are then made available within the new interface and will be placed before the new functions. This is useful in order to do basic inheritance of objects.
Arrays can be passed as parameters using the Array keyword, lists using the List keyword and maps using the Map keyword.
A return type may be defined in the interface declaration by adding the type after the method.
SizeOf may be used with Interfaces in order to get the size of the interface and OffsetOf may be used to retrieve the index of the specified function.
The pseudotypes may be used for the parameters of the functions, but not for the return value.
Note: The concept of objects, and the capability provided within PureBasic for their use, has been developed for, and mainly targeted towards, experienced programmers. However, an understanding of these concepts and capabilities are in no way a prerequisite for creating professional software or games.

Example: Basic example of object call

```
1  ; In order to access an external object (within a DLL for example),
2  ; the objects' interface must first be declared:
3
4  Interface MyObject
5    Move(x,y)
6    MoveF(x.f,y.f)
7    Destroy()
8  EndInterface
```

```
9
10  ; CreateObject is the function which creates the object, from the DLL,
11  ; whose interface has just been defined.
12  ; Create the first object...
13  ;
14  Object1.MyObject = MyCreateObject()
15
16  ; And the second one.
17  ;
18  Object2.MyObject = MyCreateObject()
19
20  ; Then the functions which have just been defined, may
21  ; be used, in order to act upon the desired object.
22  ;
23  Object1\Move(10, 20)
24  Object1\Destroy()
25
26  Object2\MoveF(10.5, 20.1)
27  Object2\Destroy()
```

Example: Example with 'Extends'

```
1   ; Define a basic Cube interface object.
2   ;
3   Interface Cube
4     GetPosition()
5     SetPosition(x)
6     GetWidth()
7     SetWidth(Width)
8   EndInterface
9
10  Interface ColoredCube Extends Cube
11    GetColor()
12    SetColor(Color)
13  EndInterface
14
15  Interface TexturedCube Extends Cube
16    GetTexture()
17    SetTexture(TextureID)
18  EndInterface
19
20  ; The interfaces for 3 different objects have now been defined, these
     objects include:
21  ;
22  ; - 'Cube' which has the: Get/SetPosition() and Get/SetWidth()
     functions.
23  ; - 'ColoredCube' which has the: Get/SetPosition(), Get/SetWidth()
     and Get/SetColor() functions.
24  ; - 'TexturedCube' which has the: Get/SetPosition(), Get/SetWidth()
     and Get/SetTexture() functions.
25  ;
```

Chapter 42

Licenses for the PureBasic applications (without using 3D engine)

```
This program makes use of the following components:

----------------------------------------------------------------------
Component: MD5
----------------------------------------------------------------------

Copyright (C) 1991-2, RSA Data Security, Inc. Created 1991. All
rights reserved.

License to copy and use this software is granted provided that it
is identified as the "RSA Data Security, Inc. MD5 Message-Digest
Algorithm" in all material mentioning or referencing this software
or this function.

License is also granted to make and use derivative works provided
that such works are identified as "derived from the RSA Data
Security, Inc. MD5 Message-Digest Algorithm" in all material
mentioning or referencing the derived work.

RSA Data Security, Inc. makes no representations concerning either
the merchantability of this software or the suitability of this
software for any particular purpose. It is provided "as is"
without express or implied warranty of any kind.

These notices must be retained in any copies of any part of this
documentation and/or software.

----------------------------------------------------------------------
Component: AES
----------------------------------------------------------------------

Optimized ANSI C code for the Rijndael cipher (now AES)

@authorVincent Rijmen <vincent.rijmen@esat.kuleuven.ac.be>
@authorAntoon Bosselaers <antoon.bosselaers@esat.kuleuven.ac.be>
@authorPaulo Barreto <paulo.barreto@terra.com.br>

This code is hereby placed in the public domain.
```

THIS SOFTWARE IS PROVIDED BY THE AUTHORS ''AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; *LOSS OF USE, DATA, OR PROFITS; OR* BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Component: SHA1

SHA-1 in C
By Steve Reid <steve@edmweb.com>
100% Public Domain

Component: zlib

Copyright (C) 1995-2012 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; *you must not* claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly — jloup@gzip.org
Mark Adler — madler@alumni.caltech.edu

Component: libpq

Portions Copyright (c) 1996-2011, PostgreSQL Global Development Group
Portions Copyright (c) 1994, The Regents of the University of California

Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without a written agreement
is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies.

```
IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY
   FOR
DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES,
   INCLUDING
LOST PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS
DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF
   THE
POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE PROVIDED HEREUNDER
   IS
ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO
   OBLIGATIONS TO
PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

--------------------------------------------------------------------
Component: sqlite3
--------------------------------------------------------------------

The author disclaims copyright to this source code.  In place of
a legal notice, here is a blessing:

    May you do good and not evil.
    May you find forgiveness for yourself and forgive others.
    May you share freely, never taking more than you give.

--------------------------------------------------------------------
Component: libjpeg
--------------------------------------------------------------------

The authors make NO WARRANTY or representation, either express or
   implied,
with respect to this software, its quality, accuracy, merchantability,
   or
fitness for a particular purpose.  This software is provided "AS IS",
   and you,
its user, assume the entire risk as to its quality and accuracy.

This software is copyright (C) 1991-2012, Thomas G. Lane, Guido
   Vollbeding.
All Rights Reserved except as specified below.

Permission is hereby granted to use, copy, modify, and distribute this
software (or portions thereof) for any purpose, without fee, subject to
   these
conditions:
(1) If any part of the source code for this software is distributed,
   then this
README file must be included, with this copyright and no-warranty notice
unaltered; and any additions, deletions, or changes to the original
   files
must be clearly indicated in accompanying documentation.
(2) If only executable code is distributed, then the accompanying
documentation must state that "this software is based in part on the
   work of
the Independent JPEG Group".
```

(3) Permission for use of this software is granted only if the user accepts
full responsibility for any undesirable consequences; the authors accept
NO LIABILITY for damages of any kind.

These conditions apply to any software derived from or based on the IJG code,
not just to the unmodified library. If you use our work, you ought to
acknowledge us.

Permission is NOT granted for the use of any IJG author's name or company name
in advertising or publicity relating to this software or products derived from
it. This software may be referred to only as "the Independent JPEG Group's
software".

We specifically permit and encourage the use of this software as the basis of
commercial products, provided that all warranty or liability claims are
assumed by the product vendor.

Component: libpng
---

libpng versions 1.2.6, August 15, 2004, through 1.5.12, July 11, 2012, are
Copyright (c) 2004, 2006-2012 Glenn Randers-Pehrson, and are
distributed according to the same disclaimer and license as libpng-1.2.5
with the following individual added to the list of Contributing Authors:

   Cosmin Truta

libpng versions 1.0.7, July 1, 2000, through 1.2.5, October 3, 2002, are
Copyright (c) 2000-2002 Glenn Randers-Pehrson, and are
distributed according to the same disclaimer and license as libpng-1.0.6
with the following individuals added to the list of Contributing Authors:

   Simon-Pierre Cadieux
   Eric S. Raymond
   Gilles Vollant

and with the following additions to the disclaimer:

   There is no warranty against interference with your enjoyment of the
   library or against infringement. There is no warranty that our
   efforts or the library will fulfill any of your particular purposes
   or needs. This library is provided with all faults, and the entire
   risk of satisfactory quality, performance, accuracy, and effort is with
   the user.

libpng versions 0.97, January 1998, through 1.0.6, March 20, 2000, are
Copyright (c) 1998, 1999, 2000 Glenn Randers-Pehrson, and are
distributed according to the same disclaimer and license as libpng-0.96,
with the following individuals added to the list of Contributing

```
   Authors:

   Tom Lane
   Glenn Randers-Pehrson
   Willem van Schaik

libpng versions 0.89, June 1996, through 0.96, May 1997, are
Copyright (c) 1996, 1997 Andreas Dilger
Distributed according to the same disclaimer and license as libpng-0.88,
with the following individuals added to the list of Contributing
   Authors:

   John Bowler
   Kevin Bracey
   Sam Bushell
   Magnus Holmgren
   Greg Roelofs
   Tom Tanner

libpng versions 0.5, May 1995, through 0.88, January 1996, are
Copyright (c) 1995, 1996 Guy Eric Schalnat, Group 42, Inc.

For the purposes of this copyright and license, "Contributing Authors"
is defined as the following set of individuals:

   Andreas Dilger
   Dave Martindale
   Guy Eric Schalnat
   Paul Schmidt
   Tim Wegner

The PNG Reference Library is supplied "AS IS".  The Contributing Authors
and Group 42, Inc. disclaim all warranties, expressed or implied,
including, without limitation, the warranties of merchantability and of
fitness for any purpose.  The Contributing Authors and Group 42, Inc.
assume no liability for direct, indirect, incidental, special,
   exemplary,
or consequential damages, which may result from the use of the PNG
Reference Library, even if advised of the possibility of such damage.

Permission is hereby granted to use, copy, modify, and distribute this
source code, or portions hereof, for any purpose, without fee, subject
to the following restrictions:

  1. The origin of this source code must not be misrepresented.

  2. Altered versions must be plainly marked as such and must not
     be misrepresented as being the original source.

  3. This Copyright notice may not be removed or altered from
     any source or altered source distribution.

The Contributing Authors and Group 42, Inc. specifically permit, without
fee, and encourage the use of this source code as a component to
supporting the PNG file format in commercial products.  If you use this
source code in a product, acknowledgment is not required but would be
appreciated.

------------------------------------------------------------------------
```

```
Component: OpenJPEG
-----------------------------------------------------------------------

Copyright (c) 2002-2007, Communications and Remote Sensing Laboratory,
   Universite catholique de Louvain (UCL), Belgium
Copyright (c) 2002-2007, Professor Benoit Macq
Copyright (c) 2001-2003, David Janssens
Copyright (c) 2002-2003, Yannick Verschueren
Copyright (c) 2003-2007, Francois-Olivier Devaux and Antonin Descampe
Copyright (c) 2005, Herve Drolon, FreeImage Team
Copyright (c) 2006-2007, Parvatha Elangovan
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS 'AS
   IS'
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
   THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
   PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS
   BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
   THE
POSSIBILITY OF SUCH DAMAGE.

-----------------------------------------------------------------------
Component: libtiff
-----------------------------------------------------------------------

Copyright (c) 1988-1997 Sam Leffler
Copyright (c) 1991-1997 Silicon Graphics, Inc.

Permission to use, copy, modify, distribute, and sell this software and
its documentation for any purpose is hereby granted without fee,
   provided
that (i) the above copyright notices and this permission notice appear
   in
all copies of the software and related documentation, and (ii) the
   names of
Sam Leffler and Silicon Graphics may not be used in any advertising or
publicity relating to the software without the specific, prior written
permission of Sam Leffler and Silicon Graphics.

THE SOFTWARE IS PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND,
EXPRESS, IMPLIED OR OTHERWISE, INCLUDING WITHOUT LIMITATION, ANY
```

```
WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

IN NO EVENT SHALL SAM LEFFLER OR SILICON GRAPHICS BE LIABLE FOR
ANY SPECIAL, INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND,
OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER OR NOT ADVISED OF THE POSSIBILITY OF DAMAGE, AND ON ANY THEORY
    OF
LIABILITY, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.

------------------------------------------------------------------------
Component: libmodplug (Module)
------------------------------------------------------------------------

This source code is public domain.

------------------------------------------------------------------------
Component: OnError
------------------------------------------------------------------------

Copyright (c) 2003 Siegfried Rings and Sebastian Lackner
All rights reserved.

------------------------------------------------------------------------
Component: udis86 (OnError)
------------------------------------------------------------------------

Copyright (c) 2002-2009 Vivek Thampi
All rights reserved.

Redistribution and use in source and binary forms, with or without
    modification,
are permitted provided that the following conditions are met:

    * Redistributions of source code must retain the above copyright
    notice,
      this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above copyright
    notice,
      this list of conditions and the following disclaimer in the
    documentation
      and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
    IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
    IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
    LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
    DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
    SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED
    AND ON
ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
    OF THIS
```

SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Component: brieflz

Copyright (c) 2002-2004 by Joergen Ibsen / Jibz
All Rights Reserved

http://www.ibsensoftware.com/

This software is provided 'as-is', without any express or implied
warranty. In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must
   not claim that you wrote the original software. If you use this
   software in a product, an acknowledgment in the product
   documentation would be appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must
   not be misrepresented as being the original software.

3. This notice may not be removed or altered from any source
   distribution.

Component: jcalg1

This software is provided as-is, without warranty of ANY KIND, either
   expressed
or implied, including but not limited to the implied warranties of
   merchantability
and/or fitness for a particular purpose. The author shall NOT be held
   liable for
ANY damage to you, your computer, or to anyone or anything else, that
   may result
from its use, or misuse. Basically, you use it at YOUR OWN RISK.

Component: lzma

LZMA SDK is written and placed in the public domain by Igor Pavlov.

Some code in LZMA SDK is based on public domain code from another
   developers:
  1) PPMd var.H (2001): Dmitry Shkarin
  2) SHA-256: Wei Dai (Crypto++ library)

Component: libzip

Component: pcre

PCRE LICENCE

The basic library functions are written in C and are freestanding. Also
included in the distribution is a set of C++ wrapper functions, and a
just-in-time compiler that can be used to optimize pattern matching.
   These
are both optional features that can be omitted when the library is
   built.

THE BASIC LIBRARY FUNCTIONS

Written by: Philip Hazel

Email local part: ph10
Email domain: cam.ac.uk

University of Cambridge Computing Service,
Cambridge, England.

PCRE JUST-IN-TIME COMPILATION SUPPORT

Written by: Zoltan Herczeg
Email local part: hzmester
Email domain: freemail.hu

STACK-LESS JUST-IN-TIME COMPILER

Written by: Zoltan Herczeg
Email local part: hzmester
Email domain: freemail.hu

THE C++ WRAPPER FUNCTIONS

Contributed by: Google Inc.

THE "BSD" LICENCE

```
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
    IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
    THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
    PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
    THE
POSSIBILITY OF SUCH DAMAGE.

-------------------------------------------------------------------------
Component: scintilla
-------------------------------------------------------------------------

License for Scintilla and SciTE

Copyright 1998-2003 by Neil Hodgson <neilh@scintilla.org>

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation.

NEIL HODGSON DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL NEIL HODGSON BE LIABLE FOR ANY
SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER
TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
OR PERFORMANCE OF THIS SOFTWARE.

-------------------------------------------------------------------------
Component: expat
-------------------------------------------------------------------------

Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                                    and Clark Cooper
Copyright (c) 2001, 2002, 2003, 2004, 2005, 2006 Expat maintainers.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
```

in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Component: miniaudio

Copyright 2023 David Reid

Permission is hereby granted, free of charge, to any person obtaining a copy of
this software and associated documentation files (the "Software"), to deal in
the Software without restriction, including without limitation the rights to
use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies
of the Software, and to permit persons to whom the Software is furnished to do
so.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.

Component: libogg

Copyright (c) 2002, Xiph.org Foundation

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

- Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

- Neither the name of the Xiph.org Foundation nor the names of its
contributors may be used to endorse or promote products derived from

this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE FOUNDATION
OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Component: libvorbis

Copyright (c) 2002-2004 Xiph.org Foundation

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

- Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

- Neither the name of the Xiph.org Foundation nor the names of its
contributors may be used to endorse or promote products derived from
this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE FOUNDATION
OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Component: neuquant

NeuQuant Neural-Net Quantization Algorithm Interface

Copyright (c) 1994 Anthony Dekker

NEUQUANT Neural-Net quantization algorithm by Anthony Dekker, 1994.
See "Kohonen neural networks for optimal colour quantization"
in "Network: Computation in Neural Systems" Vol. 5 (1994) pp 351-367.

for a discussion of the algorithm.
See also http://members.ozemail.com.au/~dekker/NEUQUANT.HTML

Any party obtaining a copy of these files from the author, directly or
indirectly, is granted, free of charge, a full and unrestricted
   irrevocable,
world-wide, paid up, royalty-free, nonexclusive right and license to
   deal
in this software and documentation files (the "Software"), including
   without
limitation the rights to use, copy, modify, merge, publish, distribute,
   sublicense,
and/or sell copies of the Software, and to permit persons who receive
copies from any such party to do so, with the only requirement being
that this copyright notice remain intact.

Modified to quantize 32bit RGBA images for the pngnq program.
Also modified to accept a number of colors argument.
Copyright (c) Stuart Coyle 2004-2006

Rewritten by Kornel Lesinski (2009)
Euclidean distance, color matching dependent on alpha channel
and with gamma correction. code refreshed for modern
   compilers/architectures:
ANSI C, floats, removed pointer tricks and used arrays and structs.

Component: libmariadb

GNU LESSER GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright Â© 2007 Free Software Foundation, Inc. <https://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this
   license
document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the
   terms and
conditions of version 3 of the GNU General Public License, supplemented
   by the
additional permissions listed below. 0. Additional Definitions.

As used herein, â this Licenseâ refers to version 3 of the GNU
   Lesser General
Public License, and the â GNU GPLâ refers to version 3 of the GNU
   General Public
License.

â The Libraryâ refers to a covered work governed by this License,
   other than an
Application or a Combined Work as defined below.

An â Applicationâ is any work that makes use of an interface provided by the
Library, but which is not otherwise based on the Library. Defining a subclass of
a class defined by the Library is deemed a mode of using an interface provided
by the Library.

A â Combined Workâ is a work produced by combining or linking an Application with
the Library. The particular version of the Library with which the Combined Work
was made is also called the â Linked Versionâ .

The â Minimal Corresponding Sourceâ for a Combined Work means the Corresponding
Source for the Combined Work, excluding any source code for portions of the
Combined Work that, considered in isolation, are based on the Application, and
not on the Linked Version.

The â Corresponding Application Codeâ for a Combined Work means the object code
and/or source code for the Application, including any data and utility programs
needed for reproducing the Combined Work from the Application, but excluding the
System Libraries of the Combined Work. 1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without
being bound by section 3 of the GNU GPL. 2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility
refers to a function or data to be supplied by an Application that uses the
facility (other than as an argument passed when the facility is invoked), then
you may convey a copy of the modified version:

    a) under this License, provided that you make a good faith effort to ensure
    that, in the event an Application does not supply the function or data, the
    facility still operates, and performs whatever part of its purpose remains
    meaningful, or b) under the GNU GPL, with none of the additional permissions
    of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header
file that is part of the Library. You may convey such object code under terms of

your choice, provided that, if the incorporated material is not limited to
numerical parameters, data structure layouts and accessors, or small macros,
inline functions and templates (ten or fewer lines in length), you do both of
the following:

a) Give prominent notice with each copy of the object code that the Library
is used in it and that the Library and its use are covered by this License.
b) Accompany the object code with a copy of the GNU GPL and this license
document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together,
effectively do not restrict modification of the portions of the Library
contained in the Combined Work and reverse engineering for debugging such
modifications, if you also do each of the following:

a) Give prominent notice with each copy of the Combined Work that the
Library is used in it and that the Library and its use are covered by this
License. b) Accompany the Combined Work with a copy of the GNU GPL and this
license document. c) For a Combined Work that displays copyright notices
during execution, include the copyright notice for the Library among these
notices, as well as a reference directing the user to the copies of the GNU
GPL and this license document. d) Do one of the following: 0) Convey the
Minimal Corresponding Source under the terms of this License, and the
Corresponding Application Code in a form suitable for, and under terms that
permit, the user to recombine or relink the Application with a modified
version of the Linked Version to produce a modified Combined Work, in the
manner specified by section 6 of the GNU GPL for conveying Corresponding
Source. 1) Use a suitable shared library mechanism for linking with the
Library. A suitable mechanism is one that (a) uses at run time a copy of the
Library already present on the user's computer system, and (b) will operate
properly with a modified version of the Library that is interface-compatible
with the Linked Version. e) Provide Installation Information, but only if

you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License. b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License â or any later versionâ applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public

License, you may choose any version of the GNU Lesser General Public License
ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether
future versions of the GNU Lesser General Public License shall apply, that
proxy's public statement of acceptance of any version is permanent authorization
for you to choose that version for the Library.

Chapter 43

Licenses for the 3D engine integrated with PureBasic

```
This program makes use of the following components:

----------------------------------------------------------------------
Component: OGRE
----------------------------------------------------------------------

OGRE (www.ogre3d.org) is made available under the MIT License.

Copyright (c) 2000-2012 Torus Knot Software Ltd

Permission is hereby granted, free of charge, to any person obtaining a
    copy
of this software and associated documentation files (the "Software"),
    to deal
in the Software without restriction, including without limitation the
    rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or
    sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included
    in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
    OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
    THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
    FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
    IN
THE SOFTWARE.

----------------------------------------------------------------------
Component: CEGUI
```

```
------------------------------------------------------------------------

Copyright (C) 2004 - 2006 Paul D Turner & The CEGUI Development Team

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS BE LIABLE FOR ANY CLAIM, DAMAGES OR
OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
OTHER DEALINGS IN THE SOFTWARE.

------------------------------------------------------------------------
Component: bullet
------------------------------------------------------------------------

Copyright (c) 2003-2010 Erwin Coumans
   http://continuousphysics.com/Bullet/

This software is provided 'as-is', without any express or implied
   warranty.
In no event will the authors be held liable for any damages arising
   from the use of this software.
Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
   freely,
subject to the following restrictions:

1. The origin of  this software must not  be misrepresented; you must
   not claim
that you wrote the original software. If you use this software in a
   product,  an
acknowledgment in  the product  documentation would  be appreciated
   but is  not
required.

2. Altered source  versions must be plainly  marked as such, and  must
not be misrepresented as being the original software.

3. This notice may not be removed or altered from any source
   distribution.

------------------------------------------------------------------------
Component: FreeImage
------------------------------------------------------------------------

FreeImage Public License - Version 1.0
```

1. Definitions.

1.1. "Contributor" means each entity that creates or contributes to the creation
of Modifications.

1.2. "Contributor Version" means the combination of the Original Code, prior
Modifications used by a Contributor, and the Modifications made by that
particular Contributor.

1.3. "Covered Code" means the Original Code or Modifications or the combination
of the Original Code and Modifications, in each case including portions thereof.

1.4. "Electronic Distribution Mechanism" means a mechanism generally accepted in
the software development community for the electronic transfer of data.

1.5. "Executable" means Covered Code in any form other than Source Code.

1.6. "Initial Developer" means the individual or entity identified as the
Initial Developer in the Source Code notice required by Exhibit A.

1.7. "Larger Work" means a work which combines Covered Code or portions thereof
with code not governed by the terms of this License.

1.8. "License" means this document.

1.9. "Modifications" means any addition to or deletion from the substance or
structure of either the Original Code or any previous Modifications. When
Covered Code is released as a series of files, a Modification is:

A. Any addition to or deletion from the contents of a file containing Original
Code or previous Modifications.

B. Any new file that contains any part of the Original Code or previous
Modifications.

1.10. "Original Code" means Source Code of computer software code which is
described in the Source Code notice required by Exhibit A as Original Code, and
which, at the time of its release under this License is not already Covered Code
governed by this License.

1.11. "Source Code" means the preferred form of the Covered Code for making
modifications to it, including all modules it contains, plus any associated

interface definition files, scripts used to control compilation and installation
of an Executable, or a list of source code differential comparisons against
either the Original Code or another well known, available Covered Code of the
Contributor's choice. The Source Code can be in a compressed or archival form,
provided the appropriate decompression or de-archiving software is widely
available for no charge.

1.12. "You" means an individual or a legal entity exercising rights under, and
complying with all of the terms of, this License or a future version of this
License issued under Section 6.1. For legal entities, "You" includes any entity
which controls, is controlled by, or is under common control with You. For
purposes of this definition, "control" means (a) the power, direct or indirect,
to cause the direction or management of such entity, whether by contract or
otherwise, or (b) ownership of fifty percent (50%) or more of the outstanding
shares or beneficial ownership of such entity.

2. Source Code License.

2.1. The Initial Developer Grant. The Initial Developer hereby grants You a
world-wide, royalty-free, non-exclusive license, subject to third party
intellectual property claims:

(a) to use, reproduce, modify, display, perform, sublicense and distribute the
Original Code (or portions thereof) with or without Modifications, or as part of
a Larger Work; and

(b) under patents now or hereafter owned or controlled by Initial Developer, to
make, have made, use and sell ("Utilize") the Original Code (or portions
thereof), but solely to the extent that any such patent is reasonably necessary
to enable You to Utilize the Original Code (or portions thereof) and not to any
greater extent that may be necessary to Utilize further Modifications or
combinations.

2.2. Contributor Grant. Each Contributor hereby grants You a world-wide,
royalty-free, non-exclusive license, subject to third party intellectual
property claims:

(a) to use, reproduce, modify, display, perform, sublicense and distribute the
Modifications created by such Contributor (or portions thereof) either on an
unmodified basis, with other Modifications, as Covered Code or as part of a
Larger Work; and

(b) under patents now or hereafter owned or controlled by Contributor, to
Utilize the Contributor Version (or portions thereof), but solely to the extent
that any such patent is reasonably necessary to enable You to Utilize the
Contributor Version (or portions thereof), and not to any greater extent that
may be necessary to Utilize further Modifications or combinations.

3. Distribution Obligations.

3.1. Application of License. The Modifications which You create or to which You
contribute are governed by the terms of this License, including without
limitation Section 2.2. The Source Code version of Covered Code may be
distributed only under the terms of this License or a future version of this
License released under Section 6.1, and You must include a copy of this License
with every copy of the Source Code You distribute. You may not offer or impose
any terms on any Source Code version that alters or restricts the applicable
version of this License or the recipients' rights hereunder. However, You may
include an additional document offering the additional rights described in
Section 3.5.

3.2. Availability of Source Code. Any Modification which You create or to which
You contribute must be made available in Source Code form under the terms of
this License either on the same media as an Executable version or via an
accepted Electronic Distribution Mechanism to anyone to whom you made an
Executable version available; *and if made available via Electronic Distribution*
Mechanism, must remain available for at least twelve (12) months after the date
it initially became available, or at least six (6) months after a subsequent
version of that particular Modification has been made available to such
recipients. You are responsible for ensuring that the Source Code version

remains available even if the Electronic Distribution Mechanism is maintained by
a third party.

3.3. Description of Modifications. You must cause all Covered Code to which you
contribute to contain a file documenting the changes You made to create that
Covered Code and the date of any change. You must include a prominent statement
that the Modification is derived, directly or indirectly, from Original Code
provided by the Initial Developer and including the name of the Initial
Developer in (a) the Source Code, and (b) in any notice in an Executable version
or related documentation in which You describe the origin or ownership of the
Covered Code.

3.4. Intellectual Property Matters

(a) Third Party Claims. If You have knowledge that a party claims an
intellectual property right in particular functionality or code (or its
utilization under this License), you must include a text file with the source
code distribution titled "LEGAL" which describes the claim and the party making
the claim in sufficient detail that a recipient will know whom to contact. If
you obtain such knowledge after You make Your Modification available as
described in Section 3.2, You shall promptly modify the LEGAL file in all copies
You make available thereafter and shall take other steps (such as notifying
appropriate mailing lists or newsgroups) reasonably calculated to inform those
who received the Covered Code that new knowledge has been obtained.

(b) Contributor APIs. If Your Modification is an application programming
interface and You own or control patents which are reasonably necessary to
implement that API, you must also include this information in the LEGAL file.

3.5. Required Notices. You must duplicate the notice in Exhibit A in each file
of the Source Code, and this License in any documentation for the Source Code,
where You describe recipients' rights relating to Covered Code. If You created
one or more Modification(s), You may add your name as a Contributor to the
notice described in Exhibit A. If it is not possible to put such notice in a

particular Source Code file due to its structure, then you must include such
notice in a location (such as a relevant directory file) where a user would be
likely to look for such a notice. You may choose to offer, and to charge a fee
for, warranty, support, indemnity or liability obligations to one or more
recipients of Covered Code. However, You may do so only on Your own behalf, and
not on behalf of the Initial Developer or any Contributor. You must make it
absolutely clear than any such warranty, support, indemnity or liability
obligation is offered by You alone, and You hereby agree to indemnify the
Initial Developer and every Contributor for any liability incurred by the
Initial Developer or such Contributor as a result of warranty, support,
indemnity or liability terms You offer.

3.6. Distribution of Executable Versions. You may distribute Covered Code in
Executable form only if the requirements of Section 3.1-3.5 have been met for
that Covered Code, and if You include a notice stating that the Source Code
version of the Covered Code is available under the terms of this License,
including a description of how and where You have fulfilled the obligations of
Section 3.2. The notice must be conspicuously included in any notice in an
Executable version, related documentation or collateral in which You describe
recipients' rights relating to the Covered Code. You may distribute the
Executable version of Covered Code under a license of Your choice, which may
contain terms different from this License, provided that You are in compliance
with the terms of this License and that the license for the Executable version
does not attempt to limit or alter the recipient's rights in the Source Code
version from the rights set forth in this License. If You distribute the
Executable version under a different license You must make it absolutely clear
that any terms which differ from this License are offered by You alone, not by
the Initial Developer or any Contributor. You hereby agree to indemnify the
Initial Developer and every Contributor for any liability incurred by the
Initial Developer or such Contributor as a result of any such terms You offer.

3.7. Larger Works. You may create a Larger Work by combining Covered Code with
other code not governed by the terms of this License and distribute the Larger
Work as a single product. In such a case, You must make sure the requirements of
this License are fulfilled for the Covered Code.

4. Inability to Comply Due to Statute or Regulation.

If it is impossible for You to comply with any of the terms of this License with
respect to some or all of the Covered Code due to statute or regulation then You
must: (a) comply with the terms of this License to the maximum extent possible;
and (b) describe the limitations and the code they affect. Such description must
be included in the LEGAL file described in Section 3.4 and must be included with
all distributions of the Source Code. Except to the extent prohibited by statute
or regulation, such description must be sufficiently detailed for a recipient of
ordinary skill to be able to understand it.

5. Application of this License.

This License applies to code to which the Initial Developer has attached the
notice in Exhibit A, and to related Covered Code.

6. Versions of the License.

6.1. New Versions. Floris van den Berg may publish revised and/or new versions
of the License from time to time. Each version will be given a distinguishing
version number.

6.2. Effect of New Versions. Once Covered Code has been published under a
particular version of the License, You may always continue to use it under the
terms of that version. You may also choose to use such Covered Code under the
terms of any subsequent version of the License published by Floris van den Berg
No one other than Floris van den Berg has the right to modify the terms
applicable to Covered Code created under this License.

6.3. Derivative Works. If you create or use a modified version of this License
(which you may only do in order to apply it to code which is not already Covered
Code governed by this License), you must (a) rename Your license so that the
phrases "FreeImage", ‘FreeImage Public License", "FIPL", or any

confusingly
similar phrase do not appear anywhere in your license and (b) otherwise make it
clear that your version of the license contains terms which differ from the
FreeImage Public License. (Filling in the name of the Initial Developer,
Original Code or Contributor in the notice described in Exhibit A shall not of
themselves be deemed to be modifications of this License.)

7. DISCLAIMER OF WARRANTY.

COVERED CODE IS PROVIDED UNDER THIS LICENSE ON AN "AS IS" BASIS, WITHOUT
WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, WITHOUT
LIMITATION, WARRANTIES THAT THE COVERED CODE IS FREE OF DEFECTS, MERCHANTABLE,
FIT FOR A PARTICULAR PURPOSE OR NON-INFRINGING. THE ENTIRE RISK AS TO THE
QUALITY AND PERFORMANCE OF THE COVERED CODE IS WITH YOU. SHOULD ANY COVERED CODE
PROVE DEFECTIVE IN ANY RESPECT, YOU (NOT THE INITIAL DEVELOPER OR ANY OTHER
CONTRIBUTOR) ASSUME THE COST OF ANY NECESSARY SERVICING, REPAIR OR CORRECTION.
THIS DISCLAIMER OF WARRANTY CONSTITUTES AN ESSENTIAL PART OF THIS LICENSE. NO
USE OF ANY COVERED CODE IS AUTHORIZED HEREUNDER EXCEPT UNDER THIS DISCLAIMER.

8. TERMINATION.

This License and the rights granted hereunder will terminate automatically if
You fail to comply with terms herein and fail to cure such breach within 30 days
of becoming aware of the breach. All sublicenses to the Covered Code which are
properly granted shall survive any termination of this License. Provisions
which, by their nature, must remain in effect beyond the termination of this
License shall survive.

9. LIMITATION OF LIABILITY.

UNDER NO CIRCUMSTANCES AND UNDER NO LEGAL THEORY, WHETHER TORT (INCLUDING
NEGLIGENCE), CONTRACT, OR OTHERWISE, SHALL THE INITIAL DEVELOPER, ANY OTHER
CONTRIBUTOR, OR ANY DISTRIBUTOR OF COVERED CODE, OR ANY SUPPLIER OF ANY OF SUCH
PARTIES, BE LIABLE TO YOU OR ANY OTHER PERSON FOR ANY INDIRECT, SPECIAL,
INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY CHARACTER INCLUDING, WITHOUT
LIMITATION, DAMAGES FOR LOSS OF GOODWILL, WORK STOPPAGE, COMPUTER

FAILURE OR
MALFUNCTION, OR ANY AND ALL OTHER COMMERCIAL DAMAGES OR LOSSES, EVEN IF SUCH
PARTY SHALL HAVE BEEN INFORMED OF THE POSSIBILITY OF SUCH DAMAGES. THIS
LIMITATION OF LIABILITY SHALL NOT APPLY TO LIABILITY FOR DEATH OR PERSONAL
INJURY RESULTING FROM SUCH PARTY'S NEGLIGENCE TO THE EXTENT APPLICABLE LAW
PROHIBITS SUCH LIMITATION. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR
LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THAT EXCLUSION AND
LIMITATION MAY NOT APPLY TO YOU.

10. U.S. GOVERNMENT END USERS.

The Covered Code is a "commercial item," as that term is defined in 48 C.F.R.
2.101 (Oct. 1995), consisting of "commercial computer software" and "commercial
computer software documentation," as such terms are used in 48 C.F.R. 12.212
(Sept. 1995). Consistent with 48 C.F.R. 12.212 and 48 C.F.R. 227.7202-1 through
227.7202-4 (June 1995), all U.S. Government End Users acquire Covered Code with
only those rights set forth herein.

11. MISCELLANEOUS.

This License represents the complete agreement concerning subject matter hereof.
If any provision of this License is held to be unenforceable, such provision
shall be reformed only to the extent necessary to make it enforceable. This
License shall be governed by Dutch law provisions (except to the extent
applicable law, if any, provides otherwise), excluding its conflict-of-law
provisions. With respect to disputes in which at least one party is a citizen
of, or an entity chartered or registered to do business in, the The Netherlands:
(a) unless otherwise agreed in writing, all disputes relating to this License
(excepting any dispute relating to intellectual property rights) shall be
subject to final and binding arbitration, with the losing party paying all costs
of arbitration; *(b) any arbitration relating to this Agreement shall be held in*
Almelo, The Netherlands; *and (c) any litigation relating to this Agreement shall*
be subject to the jurisdiction of the court of Almelo, The Netherlands with the
losing party responsible for costs, including without limitation, court costs

and reasonable attorneys fees and expenses. Any law or regulation which provides
that the language of a contract shall be construed against the drafter shall not
apply to this License.

12. RESPONSIBILITY FOR CLAIMS.

Except in cases where another Contributor has failed to comply with Section 3.4,
You are responsible for damages arising, directly or indirectly, out of Your
utilization of rights under this License, based on the number of copies of
Covered Code you made available, the revenues you received from utilizing such
rights, and other relevant factors. You agree to work with affected parties to
distribute responsibility on an equitable basis.

EXHIBIT A.

"The contents of this file are subject to the FreeImage Public License Version
1.0 (the "License"); you may not use this file except in compliance with the
License. You may obtain a copy of the License at
http://home.wxs.nl/~flvdberg/freeimage-license.txt

Software distributed under the License is distributed on an "AS IS" basis,
WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the
specific language governing rights and limitations under the License.

Component: FreeType

The FreeType Project LICENSE

2006-Jan-27

Copyright 1996-2002, 2006 by
David Turner, Robert Wilhelm, and Werner Lemberg

Introduction
============

The FreeType Project is distributed in several archive packages;
some of them may contain, in addition to the FreeType font engine,
various tools and contributions which rely on, or relate to, the
FreeType Project.

This license applies to all files found in such packages, and

which do not fall under their own explicit license. The license affects thus the FreeType font engine, the test programs, documentation and makefiles, at the very least.

This license was inspired by the BSD, Artistic, and IJG (Independent JPEG Group) licenses, which all encourage inclusion and use of free software in commercial and freeware products alike. As a consequence, its main points are that:

- We don't promise that this software works. However, we will be interested in any kind of bug reports. ('as is' distribution)
- You can use this software for whatever you want, in parts or full form, without having to pay us. ('royalty-free' usage)
- You may not pretend that you wrote this software. If you use it, or only parts of it, in a program, you must acknowledge somewhere in your documentation that you have used the FreeType code. ('credits')

We specifically permit and encourage the inclusion of this software, with or without modifications, in commercial products. We disclaim all warranties covering The FreeType Project and assume no liability related to The FreeType Project.

Finally, many people asked us for a preferred form for a credit/disclaimer to use in compliance with this license. We thus encourage you to use the following text:

```
"""
 Portions of this software are copyright © <year> The FreeType
 Project (www.freetype.org).  All rights reserved.
"""
```

Please replace <year> with the value from the FreeType version you actually use.

Legal Terms

0. Definitions

Throughout this license, the terms 'package', 'FreeType Project', and 'FreeType archive' refer to the set of files originally distributed by the authors (David Turner, Robert Wilhelm, and Werner Lemberg) as the 'FreeType Project', be they named as alpha, beta or final release.

'You' refers to the licensee, or person using the project, where 'using' is a generic term including compiling the project's source code as well as linking it to form a 'program' or 'executable'. This program is referred to as 'a program using the FreeType engine'.

This license applies to all files distributed in the original FreeType Project, including all source code, binaries and

documentation, unless otherwise stated in the file in its original, unmodified form as distributed in the original archive. If you are unsure whether or not a particular file is covered by this license, you must contact us to verify this.

The FreeType Project is copyright (C) 1996-2000 by David Turner, Robert Wilhelm, and Werner Lemberg. All rights reserved except as specified below.

1. No Warranty

THE FREETYPE PROJECT IS PROVIDED 'AS IS' WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL ANY OF THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY DAMAGES CAUSED BY THE USE OR THE INABILITY TO USE, OF THE FREETYPE PROJECT.

2. Redistribution

This license grants a worldwide, royalty-free, perpetual and irrevocable right and license to use, execute, perform, compile, display, copy, create derivative works of, distribute and sublicense the FreeType Project (in both source and object code forms) and derivative works thereof for any purpose; and to authorize others to exercise some or all of the rights granted herein, subject to the following conditions:

- Redistribution of source code must retain this license file ('FTL.TXT') unaltered; any additions, deletions or changes to the original files must be clearly indicated in accompanying documentation. The copyright notices of the unaltered, original files must be preserved in all copies of source files.

- Redistribution in binary form must provide a disclaimer that states that the software is based in part of the work of the FreeType Team, in the distribution documentation. We also encourage you to put an URL to the FreeType web page in your documentation, though this isn't mandatory.

These conditions apply to any software derived from or based on the FreeType Project, not just the unmodified files. If you use our work, you must acknowledge us. However, no fee need be paid to us.

3. Advertising

Neither the FreeType authors and contributors nor you shall use the name of the other for commercial, advertising, or promotional purposes without specific prior written permission.

We suggest, but do not require, that you use one or more of the following phrases to refer to this software in your documentation or advertising materials: 'FreeType Project', 'FreeType Engine', 'FreeType library', or 'FreeType Distribution'.

As you have not signed this license, you are not required to accept it. However, as the FreeType Project is copyrighted material, only this license, or another one contracted with the authors, grants you the right to use, distribute, and modify it. Therefore, by using, distributing, or modifying the FreeType Project, you indicate that you understand and accept all the terms of this license.

4. Contacts

There are two mailing lists related to FreeType:

- freetype@nongnu.org

  Discusses general use and applications of FreeType, as well as future and wanted additions to the library and distribution. If you are looking for support, start in this list if you haven't found anything to help you in the documentation.

- freetype-devel@nongnu.org

  Discusses bugs, as well as engine internals, design issues, specific licenses, porting, etc.

Our home page can be found at

http://www.freetype.org

Component: libogg

Copyright (c) 2002, Xiph.org Foundation

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- Neither the name of the Xiph.org Foundation nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY

THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Component: libvorbis

Copyright (c) 2002-2004 Xiph.org Foundation

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

- Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

- Neither the name of the Xiph.org Foundation nor the names of its
contributors may be used to endorse or promote products derived from
this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE FOUNDATION
OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Component: zlib

Copyright (C) 1995-2012 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty. In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not
   be
   misrepresented as being the original software.
3. This notice may not be removed or altered from any source

```
   distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu

------------------------------------------------------------------------
Component: pcre
------------------------------------------------------------------------

PCRE LICENCE
------------

PCRE is a library of functions to support regular expressions whose
   syntax
and semantics are as close as possible to those of the Perl 5 language.

Release 8 of PCRE is distributed under the terms of the "BSD" licence,
   as
specified below. The documentation for PCRE, supplied in the "doc"
directory, is distributed under the same terms as the software itself.
   The data
in the testdata directory is not copyrighted and is in the public
   domain.

The basic library functions are written in C and are freestanding. Also
included in the distribution is a set of C++ wrapper functions, and a
just-in-time compiler that can be used to optimize pattern matching.
   These
are both optional features that can be omitted when the library is
   built.


THE BASIC LIBRARY FUNCTIONS
---------------------------

Written by: Philip Hazel
Email local part: ph10
Email domain: cam.ac.uk

University of Cambridge Computing Service,
Cambridge, England.

Copyright (c) 1997-2020 University of Cambridge
All rights reserved.


PCRE JUST-IN-TIME COMPILATION SUPPORT
-------------------------------------

Written by: Zoltan Herczeg
Email local part: hzmester
Email domain: freemail.hu

Copyright(c) 2010-2020 Zoltan Herczeg
All rights reserved.


STACK-LESS JUST-IN-TIME COMPILER
--------------------------------
```

Written by: Zoltan Herczeg
Email local part: hzmester
Email domain: freemail.hu

THE C++ WRAPPER FUNCTIONS

Contributed by: Google Inc.

THE "BSD" LICENCE

Component: MeshMagick

```
Permission is hereby granted, free of charge, to any person obtaining a
    copy
of this software and associated documentation files (the "Software"),
    to deal
in the Software without restriction, including without limitation the
    rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or
    sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included
    in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
    OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
    THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
    FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
    IN
THE SOFTWARE.

-------------------------------------------------------------------------
Component: OgreBullet
-------------------------------------------------------------------------

Copyright 2007 Paul "Tuan Kuranes" Cheyrou-Lagrèze.

This file is part of OgreBullet an integration layer between the OGRE 3D
graphics engine and the Bullet physic library.

Permission is hereby granted, free of charge, to any person obtaining a
    copy
of this software and associated documentation files (the "Software"),
    to deal
in the Software without restriction, including without limitation the
    rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or
    sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included
    in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
    OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
    THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
```

```
   FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
   IN
THE SOFTWARE.

-----------------------------------------------------------------------
Component: OgreProcedural
-----------------------------------------------------------------------

This source file is part of ogre-procedural

For the latest info, see http://code.google.com/p/ogre-procedural/

Copyright (c) 2010 Michael Broutin

Permission is hereby granted, free of charge, to any person obtaining a
   copy
of this software and associated documentation files (the "Software"),
   to deal
in the Software without restriction, including without limitation the
   rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or
   sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included
   in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
   OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
   THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
   FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
   IN
THE SOFTWARE.

-----------------------------------------------------------------------
Components:
  - OpenAL
  - OgreAL
  - zziplib
  - Hydrax
-----------------------------------------------------------------------

      GNU LIBRARY GENERAL PUBLIC LICENSE
           Version 2, June 1991

 Copyright (C) 1991 Free Software Foundation, Inc.
                    675 Mass Ave, Cambridge, MA 02139, USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

[This is the first released version of the library GPL.  It is
```

numbered 2 because it goes with version 2 of the ordinary GPL.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users.

This license, the Library General Public License, applies to some specially designated Free Software Foundation software, and to any other libraries whose authors decide to use it. You can use it for your libraries, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library, or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link a program with the library, you must provide complete object files to the recipients so that they can relink them with the library, after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

Our method of protecting your rights has two steps: (1) copyright the library, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the library.

Also, for each distributor's protection, we want to make certain that everyone understands that there is no warranty for this free library. If the library is modified by someone else and passed on, we want its recipients to know that what they have is not the original version, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that companies distributing free software will individually obtain patent licenses, thus in effect transforming the program into proprietary software. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

Most GNU software, including some libraries, is covered by the ordinary
GNU General Public License, which was designed for utility programs. This
license, the GNU Library General Public License, applies to certain designated libraries. This license is quite different from the ordinary one; be sure to read it in full, and don't assume that anything in it is

the same as in the ordinary license.

  The reason we have a separate public license for some libraries is that
they blur the distinction we usually make between modifying or adding to a
program and simply using it. Linking a program with a library, without
changing the library, is in some sense simply using the library, and is
analogous to running a utility program or application program.
  However, in
a textual and legal sense, the linked executable is a combined work, a
derivative of the original library, and the ordinary General Public License
treats it as such.

  Because of this blurred distinction, using the ordinary General
Public License for libraries did not effectively promote software
sharing, because most developers did not use the libraries. We
concluded that weaker conditions might promote sharing better.

  However, unrestricted linking of non-free programs would deprive the
users of those programs of all benefit from the free status of the
libraries themselves. This Library General Public License is intended to
permit developers of non-free programs to use free libraries, while
preserving your freedom as a user of such programs to change the free
libraries that are incorporated in them. (We have not seen how to achieve
this as regards changes in header files, but we have achieved it as regards
changes in the actual functions of the Library.) The hope is that this
will lead to faster development of free libraries.

  The precise terms and conditions for copying, distribution and
modification follow. Pay close attention to the difference between a
"work based on the library" and a "work that uses the library". The
former contains code derived from the library, while the latter only
works together with the library.

  Note that it is possible for a library to be covered by the ordinary
General Public License rather than by this special one.

      GNU LIBRARY GENERAL PUBLIC LICENSE
   TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

  0. This License Agreement applies to any software library which
contains a notice placed by the copyright holder or other authorized
party saying it may be distributed under the terms of this Library
General Public License (also called "this License"). Each licensee is
addressed as "you".

  A "library" means a collection of software functions and/or data
prepared so as to be conveniently linked with application programs
(which use some of those functions and data) to form executables.

  The "Library", below, refers to any such software library or work
which has been distributed under these terms. A "work based on the
Library" means either the Library or any derivative work under
copyright law: that is to say, a work containing the Library or a

portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) The modified work must itself be a software library.

b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If

identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License.

Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also compile or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

c) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

d) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues),

conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any
particular circumstance, the balance of the section is intended to apply,
and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add
an explicit geographical distribution limitation excluding those countries,
so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Library General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our

decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Chapter 44

Macros

Syntax

```
Macro <name> [(Parameter [, ...])]
  ...
EndMacro
```

Description

Macros are a very powerful feature, mainly useful for advanced programmers. A macro is a placeholder for some code (one keyword, one line or even many lines), which will be directly inserted in the source code at the place where a macro is used. In this, it differs from procedures , as the procedures doesn't duplicate the code when they are called.
The Macro : EndMacro declaration must be done before the macro will be called for the first time. Because macros will be completely replaced by their related code at compile time, they are not local to a procedure.
A macro can not have a return type nor typed parameters. When a macro has some parameters, they are replaced in the macro code by the literal expression which is passed to the called macro. No evaluation is done as this stage, which is very important to understand: the evaluation of a line is started once all the macros found on this line are expanded.
Macros are divided into two categories: simple (without parameters) and complex (with parameters, needs the parentheses when calling it). When using no parameters, it's possible to replace any word with another word (or any expression). Macros can't be called recursively. Macro declaration can't be nested (it's not possible to define a macro inside another one).

Example: Simple macro

```
1  Macro MyNot
2    Not
3  EndMacro
4
5  a = 0
6  If MyNot a    ; Here the line will be expanded to : 'If Not a'
7    Debug "Ok"
8  EndIf
```

When using parameters, it's possible to do very flexible macros. The special concatenation character '#' can be used to create new labels or keyword by mixing the macro code and the parameter expression (spaces are not accepted between each words by the concatenation character). It's also possible to define default values for parameters, so they can be omitted when calling the macro.

Example: Macro with parameter

```
Macro UMsgBox(Title, Body)
  MessageRequester(Title, UCase(Body), 0)
EndMacro

Text$ = "World"
UMsgBox("Hello", "-"+Text$+"-") ; Here the line will be expanded like that:
                                ; 'MessageRequester("Hello", UCase("-"+Text$+"-"), 0)'
```

Example: Macro with default parameter

```
Macro UMsgBox(Title, Body = "Ha, no body specified")
  MessageRequester(Title, UCase(Body), 0)
EndMacro

UMsgBox("Hello") ; Here the line will be expanded like that:
                 ; 'MessageRequester("Hello", UCase("Ha, no body specified"), 0)'
```

Example: Macro parameter concatenation

```
Macro XCase(Type, Text)
  Type#Case(Text)
EndMacro

Debug XCase(U, "Hello")
Debug XCase(L, "Hello")
```

Example: Advanced multi-line macro

```
Macro DoubleQuote
  "
EndMacro

Macro Assert(Expression)
  CompilerIf #PB_Compiler_Debugger  ; Only enable assert in debug mode
    If Expression
      Debug "Assert (Line " + #PB_Compiler_Line + "): " + DoubleQuote#Expression#DoubleQuote
    EndIf
  CompilerEndIf
EndMacro

Assert(10 <> 10) ; Will display nothing
Assert(10 <> 15) ; Should display the assert
```

Syntax

```
UndefineMacro <name>
```

Description

UndefineMacro allows to undefine a previously defined macro, and redefine it in a different manner. Once the macro has been undefined, it is no more available for use.

Example: Undefine macro

```
1   Macro Test
2     Debug "1"
3   EndMacro
4
5   Test ; Call the macro
6
7   UndefineMacro Test ; Undefine the macro, it no more exists
8
9   Macro Test ; Now we can redefine the macro
10    Debug "2"
11  EndMacro
12
13  Test ; Call the macro
```

Syntax

```
MacroExpandedCount
```

Description

MacroExpandedCount allows to get the expanded count (number of time the macro has been expanded/called). It can be useful to generate unique identifiers in the same macro for every expansion (like label, procedure name etc.).

Example: Expanded count

```
1   Macro Test
2     Debug MacroExpandedCount
3   EndMacro
4
5   Test ; Call the macro
6   Test ; Call the macro
7   Test ; Call the macro
```

Chapter 45

Pointers and memory access

Pointers

Example

```
1  *MyScreen.Screen = OpenScreen(0, 320, 200, 8, 0)
2  mouseX = *MyScreen\MouseX  ; Assuming the Screen structure contains a
    MouseX field
```

There are only three valid methods to set the value of a pointer:
- Get the result from a function (as shown in the above example)
- Copy the value from another pointer
- Find the address of a variable, procedure or label (as shown below)
Note: unlike C/C++, in PureBasic the '*' is **always** part of the item name. Therefore '*ptr' and 'ptr' are two different variables. 'ptr' is a variable (regular one) storing a value, '*ptr' is another variable of pointer type storing an address.
Pointers and memory size
Because pointers receive only addresses as values, the memory size of a pointer is the space allowing to store an absolute address of the processor:
- On 32-bit processors the address space is limited to 32-bit, so a pointer takes 32-bit (4 bytes, like a 'long') in memory
- On 64-bit processors it takes 64-bit (8 bytes, like a 'quad') in memory, because the absolute address is on a 64-bit range.
As a consequence the type of a pointer depends of the CPU address mode, ('long' on 32-bit CPU and 'quad' on 64-bit one for example), so a pointer is a variable of type pointer.
It results from this that assigning a native type to a pointer (*Pointer.l, *Pointer.b ...) makes no sense.
Note:
- Every time a memory address needs to be stored in a variable, it should be done through a pointer. This guarantees address integrity at the compilation time whatever the CPU address mode is.
- PureBasic x86 does not generate 64-bit executables. For PureBasic programs compiled with this, the system grants them only an addressing with 32-bit pointers.
Pointers and structures
By assigning a structure to a pointer (for example *MyPointer.Point) it allows to access any memory address in a structured way (with the operator '\').

Example: Pointers and variables

```
1  Define Point1.Point, Point2.Point
2  *CurrentPoint.Point = @Point1  ; Pointer declaration, associated to a
    structure and initialized with Point1's address
```

```
  *CurrentPoint \x = 10           ; Assign value 10 to Point1\x
  *CurrentPoint.Point = @Point2   ; move to Point2's address
  *CurrentPoint \x = 20           ; Assign value 20 to Point2\x
  Debug Point1\x
  Debug Point2\x
```

Example: Pointers and array

```
  Define Point1.Point, Point2.Point
  Dim *Points.Point(1) ; 2 slots array
  *Points(0) = @Point1 ; Assign the first point variable to the first
   array slot
  *Points(1) = @Point2 ; Same for second

  *Points(0)\x = 10 ; Modify the variables through the pointers
  *Points(1)\x = 20 ;

  Debug Point1\x
  Debug Point2\x
```

Pointers allow to move, to read and to write easily in memory. Furthermore they allow programmers to reach big quantities of data without supplementary cost further to data duplication. Copying a pointer is much faster.
Pointers are also available in structures, for more information see the structures chapter .
Pointers and character strings
All variables have a permanent size in memory (2 bytes for Word, 4 bytes for a Long, etc.) except for strings variables with lengths that can change. So string variables are managed by a different way of other variables.
Thus a structure field, that makes reference to a string, store only the memory address of the string instead of the string itself: a such structure field is a pointer towards a string.

Example

```
  Text$ = "Hello"
  *Text = @Text$              ; *Text store the address of the string in
   memory
  *Pointer.String = @*Text    ; *Pointer points on *Text
  Debug *Pointer\s            ; Display the string living at the address
   stored in *Pointer (i.e. @Text$)
```

Pointers Arithmetic
Arithmetic operations on the pointers are possible and practical by using SizeOf() .

Example

```
  Dim Array.Point(1)          ; Array of points

  *Pointer.Point = @Array()   ; Store the array address
  *Pointer\x = 10             ; Change the first array element values
  *Pointer\y = 15

  *Pointer + SizeOf(Point)    ; Move to the next array element

```

```
  *Pointer\x = 7              ; Change the second array element values
  *Pointer\y = 9

  ; Display results
  For i = 0 To 1
    Debug Array(i)\x
    Debug Array(i)\y
  Next i
```

Addresses of variables

To get the address of a variable in your code, you use the at symbol (@). A common reason for using this is when you want to pass a structured type variable to a procedure . You must pass a pointer to this variable as you cannot pass structured variables directly.

Example

```
  Structure astruct
    a.w
    b.l
    c.w
  EndStructure

  Procedure SetB(*myptr.astruct)
    *myptr\b = 69
  EndProcedure

  Define.astruct myvar

  SetB(@myvar)

  Debug myvar\b
```

Addresses of literal strings

To get the address of literal string, you can use the at symbol (@) in front of it. String constants are also supported.

Example

```
  *String = @"Test"
  Debug PeekC(*String) ; Will display 84, which is the value of 'T'
```

Addresses of procedures

For advanced programmers. The most common reason to get the address of a procedure is when dealing with the OS at a low-level. Some OSes allow you to specify callback or hook functions (for some operations) which get called by the OS and allows the programmer to extend the ability of the OS routine. The address of a procedure is found in a similar way to variables .

Example

```
Procedure WindowCB(WindowID.i, Message.i, wParam.i, lParam.i)
  ; This is where the processing of your callback procedure would be performed
EndProcedure

; A special callback for the Windows OS allowing you to process window events
SetWindowCallback( @WindowCB() )
```

Addresses of labels

It can also be useful to find the address of labels in your code. This can be because you want to access the code or data stored at that label, or any other good reason you can think of. To find the address of a label, you put a question mark (?) in front of the label name.

Example

```
Debug "Size of data file = " + Str(?endofmydata - ?mydata)

DataSection
  mydata:
    IncludeBinary "somefile.bin"
  endofmydata:
EndDataSection
```

Chapter 46

Migration guide

Introduction

PureBasic is a modern programming language which evolves quickly to follow technologic changes and brings up-to-date commandset to the programmer. It sometimes involves to change or redesign some part of the language. While we try to keep these breaking changes to a minimum, it can happen and this migration guide will help to update your source from one version to another.
If you need stability instead of cutting edge features, we highly recommend to stick to 'LTS' (Long Term Support) version of PureBasic, which are released every 2 years, and are actively maintained for bug fixes.

Regular releases

migration from 5.50 to 5.60
migration from 5.40 to 5.50
migration from 5.30 to 5.40

LTS releases

migration from 5.60 to 5.72 LTS
migration from 5.20 LTS to 5.40 LTS

Chapter 47

Migration from PureBasic 5.20 LTS to 5.40 LTS

Billboard library

AddBillboard(): code change

```
; Old
AddBillboard(Billboard, BillboardGroup, x, y, z)

; New
Billboard = AddBillboard(BillboardGroup, x, y, z)
```

Cipher library

ExamineMD5Fingerprint(): code change

```
; Old
ExamineMD5Fingerprint(#FingerPrint)

; New
UseMD5FingerPrint()
StartFingerprint(#FingerPrint, #PB_Cipher_MD5)
```

ExamineSHA1Fingerprint(): code change

```
; Old
ExamineSHA1Fingerprint(#FingerPrint)

; New
UseSHA1FingerPrint()
StartFingerprint(#FingerPrint, #PB_Cipher_SHA1)
```

MD5FileFingerprint(): code change

```
; Old
Result$ = MD5FileFingerprint(Filename$)

; New
UseMD5FingerPrint()
Result$ = FileFingerprint(Filename$, #PB_Cipher_MD5)
```

MD5Fingerprint(): code change

```
; Old
Result$ = MD5Fingerprint(*Buffer, Size)

; New
UseMD5FingerPrint()
Result$ = Fingerprint(*Buffer, Size, #PB_Cipher_MD5)
```

SHA1FileFingerprint(): code change

```
; Old
Result$ = SHA1FileFingerprint(Filename$)

; New
UseSHA1FingerPrint()
Result$ = FileFingerprint(Filename$, #PB_Cipher_SHA1)
```

SHA1Fingerprint(): code change

```
; Old
Result$ = SHA1Fingerprint(*Buffer, Size)

; New
UseSHA1FingerPrint()
Result$ = Fingerprint(*Buffer, Size, #PB_Cipher_SHA1)
```

CRC32FileFingerprint(): code change

```
; Old
Result = CRC32FileFingerprint(Filename$)

; New
UseCRC32FingerPrint()
Result.l = Val("$"+FileFingerprint(Filename$, #PB_Cipher_CRC32))
```

CRC32Fingerprint(): code change

```
; Old
Result = CRC32Fingerprint(*Buffer, Size)

; New
UseCRC32FingerPrint()
Result.l = Val("$"+Fingerprint(*Buffer, Size, #PB_Cipher_CRC32))
```

NextFingerprint(): rename only

```
; Old
NextFingerprint(#FingerPrint, *Buffer, Size)

; New
AddFingerprintBuffer(#FingerPrint, *Buffer, Size)
```

Mail library

SendMail(): code change if the 'Asynchronous' parameter was used

```
; Old
SendMail(#Mail, Smtp$, Port, 1)

; New
SendMail(#Mail, Smtp$, Port, #PB_Mail_Asynchronous)
```

Packer library

RemovePackFile(): removed
PackerEntrySize(): #PB_Packer_CompressedSize support removed for ZIP and 7z archives

XML library

CreateXMLNode(): code change

```
; Old
Node = CreateXMLNode(ParentNode)
SetXMLNodeName(Node, "Name")

; New
Node = CreateXMLNode(ParentNode, "Name")
```

Screen library

AvailableScreenMemory() removed as new API doesn't support this info anymore. It was mostly returning '0' anyway.

Window library

#PB_Event_SizeWindow and #PB_Event_MoveWindow are no more realtime on Windows, use BindEvent() to get real time update.

Engine3D library

WorldCollisionAppliedImpulse() now returns a float about the applied impulse. GetX/Y/Z() are no more supported.

Various

DataSection label within Procedure are now local labels.
ASM local label prefix has been changed from "l_" to "ll_", to avoid possible clash with main labels.
#PB_LinkedList constant has been renamed to #PB_List for better consistency

Chapter 48

Migration from PureBasic 5.30 to 5.40

Cipher library

ExamineMD5Fingerprint(): code change

```
; Old
ExamineMD5Fingerprint(#FingerPrint)

; New
UseMD5FingerPrint()
StartFingerprint(#FingerPrint, #PB_Cipher_MD5)
```

ExamineSHA1Fingerprint(): code change

```
; Old
ExamineSHA1Fingerprint(#FingerPrint)

; New
UseSHA1FingerPrint()
StartFingerprint(#FingerPrint, #PB_Cipher_SHA1)
```

MD5FileFingerprint(): code change

```
; Old
Result$ = MD5FileFingerprint(Filename$)

; New
UseMD5FingerPrint()
Result$ = FileFingerprint(Filename$, #PB_Cipher_MD5)
```

MD5Fingerprint(): code change

```
; Old
Result$ = MD5Fingerprint(*Buffer, Size)

; New
UseMD5FingerPrint()
Result$ = Fingerprint(*Buffer, Size, #PB_Cipher_MD5)
```

SHA1FileFingerprint(): code change

```
; Old
```

```
  Result$ = SHA1FileFingerprint(Filename$)

  ; New
  UseSHA1FingerPrint()
  Result$ = FileFingerprint(Filename$, #PB_Cipher_SHA1)
```

SHA1Fingerprint(): code change

```
  ; Old
  Result$ = SHA1Fingerprint(*Buffer, Size)

  ; New
  UseSHA1FingerPrint()
  Result$ = Fingerprint(*Buffer, Size, #PB_Cipher_SHA1)
```

CRC32FileFingerprint(): code change

```
  ; Old
  Result = CRC32FileFingerprint(Filename$)

  ; New
  UseCRC32FingerPrint()
  Result.l = Val("$"+FileFingerprint(Filename$, #PB_Cipher_CRC32))
```

CRC32Fingerprint(): code change

```
  ; Old
  Result = CRC32Fingerprint(*Buffer, Size)

  ; New
  UseCRC32FingerPrint()
  Result.l = Val("$"+Fingerprint(*Buffer, Size, #PB_Cipher_CRC32))
```

NextFingerprint(): rename only

```
  ; Old
  NextFingerprint(#FingerPrint, *Buffer, Size)

  ; New
  AddFingerprintBuffer(#FingerPrint, *Buffer, Size)
```

Mail library

SendMail(): code change if the 'Asynchronous' parameter was used

```
  ; Old
  SendMail(#Mail, Smtp$, Port, 1)

  ; New
  SendMail(#Mail, Smtp$, Port, #PB_Mail_Asynchronous)
```

Packer library

RemovePackFile(): removed
PackerEntrySize(): #PB_Packer_CompressedSize support removed for ZIP and 7z archives

Screen library

AvailableScreenMemory() removed as new API doesn't support this info anymore. It was mostly returning '0' anyway.

Engine3D library

WorldCollisionAppliedImpulse() now returns a float about the applied impulse. GetX/Y/Z() are no more supported.

Chapter 49

Migration from PureBasic 5.40 to 5.50

Particule library

ParticleVelocity(): code changed to support current velocity.

```
; Old
ParticleVelocity(#ParticleEmitter, Minimum, Maximum)

; New
ParticleVelocity(#ParticleEmitter, Mode, Value)
```

Others

PureBasic internal manipulation of strings is now unicode only.
ASCII mode is no longer supported internally. The function *Result = Ascii(String$) allows the manipulation of ASCII strings by the user.

Chapter 50

Migration from PureBasic 5.50 to 5.60

Cipher library

Base64Encoder: function renamed

```
; Old
Base64Encoder()

; New
Base64EncoderBuffer()
```

Base64Decoder: function renamed

```
; Old
Base64Decoder()

; New
Base64DecoderBuffer()
```

Others

'Define.b' standalone syntax, to change default type for untyped variables is now forbidden and will result in a 'Syntax error'. Removing the statement will solve this (ensure to type your untyped variables accordindly). Note: 'Define.b a, b, c' is still supported.

Chapter 51

Migration from PureBasic 5.60 to 5.72 LTS

Math library

Sign(): now returns an integer instead of a float.

```
; Old
Var.d(f) = Sign(X)

; New
Var.i = Sign(X)
```

Window library

PostEvent(): now returns a result as it could fail in some extreme case

Others

In MS WIndows, the default font changed from MS Shell Dlg to Segoe UI size 9.

Chapter 52

Module

Syntax

```
DeclareModule <name>
  ...
EndDeclareModule

Module <name>
  ...
EndModule

UseModule <name>
UnuseModule <name>
```

Description

Modules are an easy way to isolate code parts from the main code, allowing code reuse and sharing without risk of name conflict. In some other programming languages, modules are known as 'namespaces'. A module must have a DeclareModule section (which is the public interface) and an associated Module section (which is the implementation). Only items declared in the DeclareModule section will be accessible from outside the module. All the code put in the Module section will be kept private to this module. Items from main code like procedures, variables etc. won't be accessible inside the module, even if they are global . A module can be seen as a blackbox, an empty code sheet where item names can not conflict with the main code. It makes it easier to write specific code, as simple names can be reused within each module without risk of conflict.
Items allowed in the DeclareModule section can be the following: procedure (only procedure declaration allowed), structure, macro, variable, constant, enumeration, array, list, map and label.
To access a module item from outside the module, the module name has to be specified followed by the '::' separator. When explicitly specifying the module name, the module item is available everywhere in the code source, even in another module. All items in the DeclareModule section can be automatically imported into another module or in the main code using UseModule. In the case of a module name conflict, the module items won't be imported and a compiler error will be raised. UnuseModule removes the module items. UseModule is not mandatory to access a module item, but the module name needs to be specified.
To share information between modules, a common module can be created and then used in every module which needs it. It's the common way have global data available for all modules.
Default items available in modules are all PureBasic commands, structure and constants. Therefore module items can not be named like internal PureBasic commands, structures or constants.
All code put inside DeclareModule and Module sections is executed like any other code when the program flow reaches the module.
When the statements Define, EnableExplicit, EnableASM are used inside a module, they have no effect outside the respective module, and vice versa.

Note: modules are not mandatory in PureBasic but are recommended when building big projects. They help to build more maintainable code, even if it is slightly more verbose than module-free code. Having a DeclareModule section make the module pretty much self-documented for use when reusing and sharing it.

Example

```
1
2   ; Every items in this sections will be available from outside
3   ;
4   DeclareModule Ferrari
5     #FerrariName$ = "458 Italia"
6
7     Declare CreateFerrari()
8   EndDeclareModule
9
10  ; Every items in this sections will be private. Every names can be
     used without conflict
11  ;
12  Module Ferrari
13
14    Global Initialized = #False
15
16    Procedure Init() ; Private init procedure
17      If Initialized = #False
18        Initialized = #True
19        Debug "InitFerrari()"
20      EndIf
21    EndProcedure
22
23    Procedure CreateFerrari()
24      Init()
25      Debug "CreateFerrari()"
26    EndProcedure
27
28  EndModule
29
30
31  Procedure Init() ; Main init procedure, doesn't conflict with the
     Ferrari Init() procedure
32    Debug "Main init()"
33  EndProcedure
34
35  Init()
36
37  Ferrari::CreateFerrari()
38  Debug Ferrari::#FerrariName$
39
40  Debug "------------------------------"
41
42  UseModule Ferrari ; Now import all public item directly in the main
     program scope
43
44  CreateFerrari()
45  Debug #FerrariName$
```

Example: With a common module

```
1
2   ; The common module, which will be used by others to share data
3   ;
4   DeclareModule Cars
5     Global NbCars = 0
6   EndDeclareModule
7
8   Module Cars
9   EndModule
10
11  ; First car module
12  ;
13  DeclareModule Ferrari
14  EndDeclareModule
15
16  Module Ferrari
17    UseModule Cars
18
19    NbCars+1
20  EndModule
21
22  ; Second car module
23  ;
24  DeclareModule Porsche
25  EndDeclareModule
26
27  Module Porsche
28    UseModule Cars
29
30    NbCars+1
31  EndModule
32
33  Debug Cars::NbCars
```

Chapter 53

NewList

Syntax

```
NewList name.<type>()
```

Description

NewList allows to declare a new dynamic list. Each element of the list is allocated dynamically. There are no element limits, so there can be as many as needed. A list can have any Variables standard or structured type. To view all commands used to manage lists, see the List library.
The new list are always locals, which means Global or Shared commands have to be used if a list declared in the main source need to be used in procedures. It is also possible to pass a list as parameter to a procedure by using the keyword List.
For fast swapping of list contents the Swap keyword is available.

Example: Simple list

```
1   NewList MyList()
2
3   AddElement(MyList())
4   MyList() = 10
5
6   AddElement(MyList())
7   MyList() = 20
8
9   AddElement(MyList())
10  MyList() = 30
11
12  ForEach MyList()
13    Debug MyList()
14  Next
```

Example: List as procedure parameter

```
1   NewList Test()
2
3   ; Adding the first 2 items to the list:
4   AddElement(Test())
5   Test() = 1
```

```
  AddElement(Test())
  Test() = 2

  Procedure DebugList(count, List ParameterList())
    ; Adding further items to the list:
    For i=1 To count
      AddElement(ParameterList())
      ParameterList() = i
    Next

    ; Display all items in the list:
    i = 1
    ForEach ParameterList()
      MessageRequester("List", "Entry" + Str(i) + " = " +
   Str(ParameterList()))
      i + 1
    Next
  EndProcedure

  ; Add another 3 items to the list and display the list contents:
  DebugList(3, Test())
```

Chapter 54

NewMap

Syntax

```
NewMap name.<type>([Slots])
```

Description

NewMap allows to declare a new map, also known as hashtable or dictionary. It allows to quickly reference an element based on a key. Each key in the map are unique, which means it can't have two distinct elements with the same key. There are no element limits, so there can be as many as needed. A map can have any standard or structured type. To view all commands used to manage maps, see the Map library.
When using a new key, a new element is automatically added to the map even without an assignment. If another element with the same key is already in the map, it will be replaced by the new one. Once an element as been accessed or created, it becomes the current element of the map, and further access to this element can be done without specify the key. This is useful when using structured map, as no more element lookup is needed to access different structure field.
New maps are always locals by default, so Global or Shared commands have to be used if a map declared in the main source need to be used in procedures. It is also possible to pass a map as parameter to a procedure by using the keyword Map.
For fast swapping of map elements the Swap keyword is available.
The optional 'Slots' parameter defines how much slots the map will have have to store its elements. The more slots is has, the faster it will be to access an element, but the more memory it will use. It's a tradeoff depending of how many elements the map will ultimately contains and how fast the random access should be. The default value is 512. This parameter has no impact about the number of elements a map can contain.

Example: Simple map

```
1   NewMap Country.s()
2
3   Country("GE") = "Germany"
4   Country("FR") = "France"
5   Country("UK") = "United Kingdom"
6
7   Debug Country("FR")
8
9   ForEach Country()
10    Debug Country()
11  Next
```

Example: Map as procedure parameter

```
NewMap Country.s()

Country("GE") = "Germany"
Country("FR") = "France"
Country("UK") = "United Kingdom"

Procedure DebugMap(Map ParameterMap.s())

  ParameterMap("US") = "United States"

  ForEach ParameterMap()
    Debug ParameterMap()
  Next

EndProcedure

DebugMap(Country())
```

Example: Structured map

```
Structure Car
  Weight.l
  Speed.l
  Price.l
EndStructure

NewMap Cars.Car()

; Here we use the current element after the new insertion
;
Cars("Ferrari F40")\Weight = 1000
Cars()\Speed = 320
Cars()\Price = 500000

Cars("Lamborghini Gallardo")\Weight = 1200
Cars()\Speed = 340
Cars()\Price = 700000

ForEach Cars()
  Debug "Car name: "+MapKey(Cars())
  Debug "Weight: "+Str(Cars()\Weight)
Next
```

Example: Item created without assignment

```
NewMap IconMap()

IconMap("1") = 1
IconMap("2") = 2

Debug FindMapElement(IconMap(), "3") ; Displays 0 because the item
 does not exist

```

```
8   If IconMap("3") : EndIf ; PureBasic creates the item without
      assignment because a new key is detected
9
10  Debug FindMapElement(IconMap(), "3") ; Displays the memory address of
      the element
```

Chapter 55

Others Commands

Syntax

```
Goto <label>
```

Description

This command is used to transfer the program directly to the labels position. Be cautious when using this function, as incorrect use could cause a program to crash...
Note: To exit a loop safely, you must always use Break instead of Goto, and never use it inside a Select/EndSelect block (unless you have the ability to correctly manage the stack yourself).

Syntax

```
End [ExitCode]
```

Description

Ends the program execution correctly. The 'ExitCode' optional parameter can be specified if the program need to returns an error code (widely used in console programs).
The 'ExitCode' can be further used e.g. with the ProgramExitCode() command.

Syntax

```
Swap <expression>, <expression>
```

Description

Swaps the value of the both expression, in an optimized way. The both <expression> have to be a variable , an array element, a list element or a map element (structured or not) and have to be one of the PureBasic native type like long (.l), quad (.q), string etc.

Example: Swapping of strings

```
Hello$ = "Hello"
World$ = "World"

Swap Hello$, World$

Debug Hello$+" "+World$
```

Example: Swapping of multi-dimensional arrays elements

```
Dim Array1(5,5)
Dim Array2(5,5)
Array1(2,2) = 10      ; set initial contents
Array2(3,3) = 20

Debug Array1(2,2) ; will print 10
Debug Array2(3,3) ; will print 20

Swap Array1(2,2) , Array2(3,3)  ; swap 2 arrays elements

Debug "Array contents after swapping:"
Debug Array1(2,2)     ; will print 20
Debug Array2(3,3)     ; will print 10
```

Chapter 56

Procedures

Syntax

```
Procedure[.<type>] name(<parameter1[.<type>]> [, <parameter2[.<type>]
    [= DefaultValue]>, ...])
  ...
  [ProcedureReturn value]
EndProcedure
```

Description

A Procedure is a part of code independent from the main code which can have any parameters and its own variables . In PureBasic, a recurrence is fully supported for the procedures and any procedure can call it itself. At each call of the procedure the variables inside will start with a value of 0 (null). To access main code variables, they have to be shared by using Shared or Global keywords (see also the Protected and Static keywords).
The last parameters can have a default value (needs to be a constant expression), so if these parameters are omitted when the procedure is called, the default value will be used.
Arrays can be passed as parameters using the Array keyword, lists using the List keyword and maps using the Map keyword.
A procedure can return a value or a string if necessary. You have to set the type after Procedure and use the ProcedureReturn keyword at any moment inside the procedure. A call of ProcedureReturn exits immediately the procedure, even when its called inside a loop.
ProcedureReturn can't be used to return an array , list or a map . For this purpose pass the array, the list or the map as parameter to the procedure.
If no value is specified for ProcedureReturn, the return value will be zero if the compiler uses the C backend and for all processors managed by PureBasic, or the content of the 'eax' register in 32 bits (X86 processors) or 'rax' in 64 bits (X64 processors) if the compiler uses the ASM back-end, (see for more information).
Note: To return strings from DLLs, see DLLs . For advanced programmers ProcedureC is available and will declare the procedure using 'CDecl' instead of 'StandardCall' calling convention. For more information on library calling standards (cdecl, stdcall, fastcall), see here.
Selected procedures can be executed asynchronously to the main program by using of threads .

Example: Procedure with a numeric variable as return-value

```
1  Procedure Maximum(nb1, nb2)
2    If nb1 > nb2
3      Result = nb1
4    Else
5      Result = nb2
```

```
    EndIf

    ProcedureReturn Result
  EndProcedure

  Result = Maximum(15, 30)
  Debug Result
```

Example: Procedure with a string as return-value

```
  Procedure.s Attach(String1$, String2$)
    ProcedureReturn String1$+" "+String2$
  EndProcedure

  Result$ = Attach("PureBasic", "Coder")
  Debug Result$
```

Example: Parameter with default value

```
  Procedure a(a, b, c=2)
    Debug c
  EndProcedure

  a(10, 12)       ; 2 will be used as default value for 3rd parameter
  a(10, 12, 15)
```

Example: List as parameter

```
  NewList Test.Point()

  AddElement(Test())
  Test()\x = 1
  AddElement(Test())
  Test()\x = 2

  Procedure DebugList(c.l, List ParameterList.Point())

    AddElement(ParameterList())
    ParameterList()\x = 3

    ForEach ParameterList()
      MessageRequester("List", Str(ParameterList()\x))
    Next

  EndProcedure

  DebugList(10, Test())
```

Example: Array as parameter

```
Dim Table.Point(10, 15)

Table(0,0)\x = 1
Table(1,0)\x = 2

Procedure TestIt(c.l, Array ParameterTable.Point(2))  ; The table support 2 dimensions

  ParameterTable(1, 2)\x = 3
  ParameterTable(2, 2)\x = 4

EndProcedure

TestIt(10, Table())

MessageRequester("Table", Str(Table(1, 2)\x))
```

Example: Static, dynamic array and passing a Structure to a Procedure

```
Structure Whatever
  a.l
  b.l[2]          ; Static array (Standard C) with 2 values b[0] and b[1], not resizable
  Array c.l(3,3)  ; Dynamic array with 16 values c(0,0) to c(3,3), resizable with ReDim()
EndStructure

MyVar.Whatever

Procedure MyProcedure(*blahblah.Whatever)
  *blahblah\a = 5
  *blahblah\b[0] = 1
  *blahblah\b[1] = 2
  *blahblah\c(3,3) = 33
EndProcedure

MyProcedure(@MyVar)
Debug MyVar\a
Debug MyVar\b[0]
Debug MyVar\b[1]
Debug MyVar\c(3,3)
```

Example: Call a function by its name

```
 Prototype Function()

Runtime Procedure Function1()
    Debug "I call Function1 by its name"
EndProcedure

Procedure LaunchProcedure(Name.s)
    Protected ProcedureName.Function = GetRuntimeInteger(Name + "()")
    ProcedureName()
EndProcedure
```

```
11
12    LaunchProcedure("Function1") ; Display "I call Function1 by its name"
```

Syntax

```
Declare[.<type>] name(<parameter1[.<type>]> [, <parameter2[.<type>] [=
    DefaultValue]>, ...])
```

Description

Sometimes a procedure need to call another procedure which isn't declared before its definition. This is annoying because the compiler will complain 'Procedure <name> not found'. Declare can help in this particular case by declaring only the header of the procedure. Nevertheless, the Declare and real Procedure declaration must be identical (including the correct type and optional parameters, if any). For advanced programmers DeclareC is available and will declare the procedure using 'CDecl' instead of 'StandardCall' calling convention.

Example

```
1    Declare Maximum(Value1, Value2)
2
3    Procedure Operate(Value)
4      Maximum(10, 2)       ; At this time, Maximum() is unknown.
5    EndProcedure
6
7    Procedure Maximum(Value1, Value2)
8      ProcedureReturn 0
9    EndProcedure
```

Chapter 57

Protected

Syntax

```
Protected[.<type>] <variable[.<type>]> [= <expression>] [, ...]
```

Description

Protected allows a variable to be accessed only in a Procedure even if the same variable has been declared as Global in the main program. Protected in its function is often known as 'Local' from other BASIC dialects. Each variable can have a default value directly assigned to it. If a type is specified after Protected, the default type is changed for this declaration. Protected can also be used with arrays , lists and maps .
The value of the local variable will be reinitialized at each procedure call. To avoid this, you can use the keyword Static , to separate global from local variables while keeping their values.

Example: With variable

```
1   Global a
2   a = 10
3
4   Procedure Change()
5     Protected a
6     a = 20
7   EndProcedure
8
9   Change()
10  Debug a ; Will print 10, as the variable has been protected.
```

Example: With array

```
1   Global Dim Array(2)
2   Array(0) = 10
3
4   Procedure Change()
5     Protected Dim Array(2) ; This array is protected, it will be local.
6     Array(0) = 20
7   EndProcedure
8
9   Change()
```

```
10 | Debug Array(0) ; Will print 10, as the array has been protected.
```

Chapter 58

Prototypes

Syntax

```
Prototype.<type> name(<parameter>, [, <parameter> [= DefaultValue]...])
```

Description

For advanced programmers. Prototype allows to declare a type which will map a function. It's useful when used with a variable, to do a function pointer (the variable value will be the address the function to call, and can be changed at will).
This feature can replace the OpenLibrary() and CallFunction() sequence as it has some advantages: type checking is done, number of of parameters is validated.
Unlike CallFunction(), it can deal with double, float and quad variables without any problem. To get easily the pointer of a library function, use GetFunction() .
The last parameters can have a default value (need to be a constant expression), so if these parameters are omitted when the function is called, the default value will be used.
By default the function will use the standard call convention (stdcall on x86, or fastcall on x64). If the function pointer will be a C one, the PrototypeC variant should be used instead. For more information on library calling standards (cdecl, stdcall, fastcall), see here.
The pseudotypes can be used for the parameters, but not for the returned value.

Example

```
1   CompilerIf #PB_Compiler_OS <> #PB_OS_Windows
2     CompilerError "This sample only works on Windows"
3   CompilerEndIf
4
5   Prototype.i ProtoMessageBox(Window.i, Body$, Title$, Flags.i = 0)
6
7   If OpenLibrary(0, "User32.dll")
8
9     ; 'MsgBox' is a variable with a 'ProtoMessageBox' type
10    ;
11    MsgBox.ProtoMessageBox = GetFunction(0, "MessageBoxW")
12
13    MsgBox(0, "Hello", "World") ; We don't specify the flags
14  EndIf
```

Example: With pseudotypes

```
CompilerIf #PB_Compiler_OS <> #PB_OS_Windows
  CompilerError "This sample only works on Windows"
CompilerEndIf

; We use the 'p-ascii' pseudotype for the string parameters, as
; MessageBoxA() is an ascii only function. The compiler will
; automatically converts the strings to unicode when needed.
;
Prototype.i ProtoMessageBoxA(Window.i, Body.p-ascii, Title.p-ascii,
 Flags.i = 0)

If OpenLibrary(0, "User32.dll")

  ; 'MsgBox' is a variable with a 'ProtoMessageBoxA' type
  ;
  MsgBox.ProtoMessageBoxA = GetFunction(0, "MessageBoxA")

  MsgBox(0, "Hello", "World") ; We don't specify the flags
EndIf
```

Chapter 59

Pseudotypes

Description

For advanced programmers. The pseudotypes are a way to ease the programming when dealing with external libraries which need datatypes that are not internally supported by PureBasic. In this case, it is possible to use the predefined pseudotype which will do the appropriate conversion on the fly, without extra work. As they are not 'real' types, the chosen naming scheme is explicitly different: a 'p-' prefix (for 'pseudo') is part of the type name. The available pseudotypes are:

```
  p-ascii: acts as a string type, but will always convert the string to
    ascii before
           calling the function, even when the program is compiled in
    unicode
mode.
           It is very useful when accessing a shared library which
    doesn't have unicode
           support in an unicode program for example.

  p-utf8: acts as a string type, but will always convert the string to
    utf8 before calling the
          function. It is very useful when accessing a shared library
    which needs its unicode

          string be passed as UTF8 instead of ascii or unicode strings.

  p-bstr: acts as a string type, but will always convert the string to
    bstr before calling the
          function. It is very useful when accessing a shared library
    which needs bstr parameters,
          like COM components.

  p-unicode: acts as a string type, but will always convert the string
    to unicode
before
            calling the function, even when the program is compiled in
    ascii mode.
            It is very useful when accessing a shared library which
    only supports unicode
            in an ascii program for example.

  p-variant: acts as a numeric type, will adjust the function call to
    use the variant parameter
```

```
            correctly. It is very useful when accessing a shared
    library which needs 'variant' parameters,
            like COM components.
```

The pseudotypes can only be used with prototypes , interfaces and imported functions. The pseudotypes does the appropriate conversion only if it is necessary.

Example

```
1   Import "User32.lib"
2
3     ; We use the 'p-ascii' pseudotype for the string parameters, as
4     ; MessageBoxA() is an ASCII only function. The compiler will
5     ; automatically converts the strings to ASCII when needed.
6     ;
7     MessageBoxA(Window.i, Body.p-ascii, Title.p-ascii, Flags.i = 0)
8
9   EndImport
10
11  ; It will work correctly even if the internal PureBasic strings are
      in unicode
12  ; because the compiler will take care of the ASCII conversion
      automatically.
13  ;
14  MessageBoxA(0, "Hello", "World")
```

Chapter 60

PureBasic objects

Introduction

The purpose of this section is to describe the behavior, creation, and handling of objects in PureBasic. For the demonstration, we will use the Image object, but the same logic applies to all other PureBasic objects. When creating an Image object, we can do it in two ways: indexed and dynamic.

I. Indexed numbering

The static, indexed way, allows you to reference an object by a predefined numeric value. The first available index number is 0 and subsequent indexes are allocated sequentially. This means that if you use the number 0 and then the number 1000, 1001 indexes will be allocated and 999 (from 1 to 999) will be unused, which is not an efficient way to use indexed objects. If you need a more flexible method, use the dynamic way of allocating objects, as described in section II. The indexed way offers several advantages:

- Easier handling, since no variables or arrays are required.
- 'Group' processing, without the need to use an intermediate array.
- Use the object in procedures without declaring anything in global (if using a constant or a number).
- An object that is associated with an index is automatically freed when reusing that index.

The maximum index number is limited to an upper bound, depending of the object type (usually from 5000 to 60000). Enumerations are strongly recommended if you plan to use sequential constants to identify your objects (which is also recommended).

Example

```
1  CreateImage(0, 640, 480) ; Create an image, the n°0
2  ResizeImage(0, 320, 240) ; Resize the n°0 image and change its
    @ReferenceLink "handles" "handle"
```

Example

```
1  CreateImage(2, 640, 480) ; Create an image, the n°2
```

```
2   ResizeImage(2, 320, 240) ; Resize the n°2 image and change its
      @ReferenceLink "handles" "handle"
3   CreateImage(2, 800, 800) ; Create a new image in the n°2 index, the
      old one is automatically free'ed
```

Example

```
1   For k = 0 To 9
2     CreateImage(k, 640, 480) ; Create 10 different images, numbered
      from 0 to 9
3     ResizeImage(k, 320, 240) ; Resize images into half width/height
      and change its @ReferenceLink "handles" "handle"
4   Next
```

Example

```
1   #ImageBackground = 0
2   #ImageButton     = 1
3
4   CreateImage(#ImageBackground, 640, 480) ; Create an image (n°0)
5   ResizeImage(#ImageBackground, 320, 240) ; Resize the background image
6   CreateImage(#ImageButton    , 800, 800) ; Create an image (n°1)
```

II. Dynamic numbering

Sometimes, indexed numbering isn't very handy to handle dynamic situations where we need to deal with an unknown number of objects. PureBasic provides an easy and complementary way to create objects in a dynamic manner. Both methods (indexed and dynamic) can be used together at the same time without any conflict. To create a dynamic object, you just have to specify the #PB_Any constant instead of the indexed number, and the dynamic number will be returned as result of the function. Then just use this number with the other object functions in the place where you would use an indexed number (except to create a new object). This way of object handling can be very useful when used in combination with a list , which is also a dynamic way of storage.

Example

```
1   DynamicImage1 = CreateImage(#PB_Any, 640, 480) ; Create a
      dynamic image
2   ResizeImage(DynamicImage1, 320, 240) ; Resize the
      DynamicImage1
```

A complete example of dynamic objects and lists can be found here:
Further description and an example of dynamic numbering multiple windows and gadgets can be found in the related 'Beginners chapter' .

Overview of the different PureBasic objects

Different PureBasic objects (windows, gadgets, sprites, etc.) can use the same range of object numbers again. So the following objects can be enumerated each one starting at 0 (or other value) and PureBasic differs them by their type:

- Database
- Dialog
- Entity
- File
- FTP
- Gadget (including the ScintillaGadget())
- Gadget3D
- Image
- Library
- Light
- Mail
- Material
- Menu (not MenuItem() , as this is no object)
- Mesh
- Movie
- Music
- Network
- Node
- Particle
- RegularExpression
- SerialPort
- Sound
- Sound3D
- Sprite
- StatusBar
- Texture
- ToolBar
- Window
- Window3D
- XML

Chapter 61

Building a PureLibrary

Introduction

PureBasic allows to easily create custom libraries to extend the core PureBasic commandset with more commands. The generated library will be located in the PureBasic/PureLibraries/UserLibraries/ folder, so be sure to have the proper access rights when trying to generate one.
For now, the purelibrary creation is only available from commandline, using the C backend compiler using the '–purelibrary' option. It's also possible to use the great library creation tool from the IDE by 'Pf Shadoko' found here which also come with some more features.
As a PureLibary adds new command to the standard commandset, the new command name must be different than the internal commands. A PureLibrary should always been compiled with the PureBasic compiler it will be used on. If a PureLibrary doesn't come with it's associated code source, it will be very likely to break in a future version of PureBasic and should be avoided.
A PureLibrary can also be created using C/C++ or ASM, please look in the PureLibrary/SDK/ folder.

Exporting functions

No code should be written outside procedures, except for object declaration.
To export a function when creating a PureLibrary, it needs to be declared as ProcedureDLL. If optional parameters are needed another function with the same name but with an incremented number can be used. A 'QuickHelp' comment can also be added to have the quick help displayed in the IDE when using the command.
Example:

```
1  ; QuickHelp MyMax(Min [, Max [, Flags, Mode]]) - A standard min/max
     function
2  ProcedureDLL MyMax3(a, b, c, d)
3    ; Your code here
4    ProcedureReturn a
5  EndProcedure
6
7  ProcedureDLL MyMax2(a, b)
8    ProcedureReturn MyMax3(a, b, 0, 0)
9  EndProcedure
10
11 ProcedureDLL MyMax(a)
12   ProcedureReturn MyMax2(a, 0)
13 EndProcedure
```

Automatic functions

There are two special functions for automatic initialization and to free the library: InitPureLibrary() and FreePureLibrary(). Unlike other library functions, these are not declared with ProcedureDLL but with Procedure. These two functions are automatically called when the program is launched and when the program is terminated.

Example:

```
Procedure InitPureLibrary()
  ; Your init routine here
EndProcedure

Procedure FreePureLibrary()
  ; Your free routine here
EndProcedure
```

Disabling a PureLibrary

When coding the PureLibrary, it might be useful to ignore the current library with the DisablePureLibrary so the function name can be used again.

Example:

```
DisablePureLibrary MyCoolLib

; All functions found in 'MyCoolLib' will be ignored, so their name
  can be used again with ProcedureDLL
```

Removing a PureLibrary

To remove a custom PureLibrary, just delete the according file in the PureBasic/PureLibraries/UserLibraries/ folder.

Chapter 62

Repeat : Until

Syntax

```
Repeat
  ...
Until <expression> [or Forever]
```

Description

This function will loop until the <expression> becomes true. The number of loops is unlimited. If an endless loop is needed then use the Forever keyword instead of Until. With the Break command, it is possible to exit the Repeat : Until loop during any iteration. With Continue command, the end of the current iteration may be skipped.

Example

```
1  a=0
2  Repeat
3    a=a+1
4  Until a>100
```

This will loop until "a" takes a value > to 100, (it will loop 101 times).

Chapter 63

Residents

Description

Residents are precompiled files which are loaded when the compiler starts. They can be found in the 'residents' folder of the PureBasic installation path. A resident file must have the extension '.res' and can contain the following items: structures , interfaces , prototypes , macros and constants . It can not contain dynamic code or procedures .
When a resident is loaded, all its content is available for the program being compiled. That's why all built-in constants like #PB_Event_CloseWindow are available, they are in the 'PureBasic.res' file. All the API structures and constants are also in a resident file. Using residents is a good way to store the common macros, structure and constants so they will be available for every programs. When distributing an user library, it's also a nice solution to provide the needed constants and structures, as PureBasic does.
To create a new resident, the command-line compiler needs to be used, as there is no option to do it from the IDE. It is often needed to use /IGNORERESIDENT and /CREATERESIDENT at the same time to avoid duplicate errors, as the previous version of the resident is loaded before creating the new one.
Residents greatly help to have a faster compilation and compiler start, as all the information is stored in binary format. It is much faster to load than parsing an include file at every compilation.

Chapter 64

Runtime

Syntax

```
Runtime Variable
Runtime #Constant
Runtime Procedure() declaration
Runtime Enumeration declaration
```

Description

For advanced programmers. Runtime is used to create runtime accessible list of programming objects like variables, constants and procedures. Once compiled a program doesn't have variable, constant or procedure label anymore as everything is converted into binary code. Runtime enforces the compiler to add an extra reference for a specific object to have it available through the Runtime library. The objects can be manipulated using their string reference, even when the program is compiled.
To illustrate the use of Runtime: the Dialog library use it to access the event procedure associated to a gadget . The name of the procedure to use for the event handler is specified in the XML file (which is text format), and then the dialog library use GetRuntimeInteger() to resolve the procedure address at runtime. It's not needed to recompile the program to change it.
Another use would be adding a small realtime scripting language to the program, allowing easy modification of exposed variables, using runtime constants values. While it could be done manually by building a map of objects, the Runtime keyword allows to do it in a standard and unified way.

Example: Procedure

```
1  Runtime Procedure OnEvent()
2    Debug "OnEvent"
3  EndProcedure
4
5  Debug GetRuntimeInteger("OnEvent()") ; Will display the procedure
     address
```

Example: Enumeration

```
1  Runtime Enumeration
2    #Constant1 = 10
3    #Constant2
4    #Constant3
5  EndEnumeration
```

```
6
7   Debug GetRuntimeInteger("#Constant1")
8   Debug GetRuntimeInteger("#Constant2")
9   Debug GetRuntimeInteger("#Constant3")
```

Example: Variable

```
1   Define a = 20
2   Runtime a
3
4   Debug GetRuntimeInteger("a")
5   SetRuntimeInteger("a", 30)
6
7   Debug a ; the variable has been modified
```

Example: Call a function by its name

```
1    Prototype Function()
2
3   Runtime Procedure Function1()
4       Debug "I call Function1 by its name"
5   EndProcedure
6
7   Runtime Procedure Function2()
8       Debug "I call Function2 by its name"
9   EndProcedure
10
11  Procedure LaunchProcedure(Name.s)
12      Protected ProcedureName.Function = GetRuntimeInteger(Name + "()")
13      ProcedureName()
14  EndProcedure
15
16  LaunchProcedure("Function1") ; Display "I call Function1 by its name"
17  LaunchProcedure("Function2") ; Display "I call Function2 by its name"
```

Chapter 65

Select : EndSelect

Syntax

```
Select <expression1>
  Case <expression> [, <expression> [<numeric expression> To <numeric
   expression>]]
     ...
  [Case <expression>]
     ...
  [Default]
     ...
EndSelect
```

Description

Select provides the ability to determine a quick choice. The program will execute the <expression1> and retain its value in memory. It will then compare this value to all of the Case <expression> values and if a given Case <expression> value is true, it will then execute the corresponding code and quit the Select structure. Case supports multi-values and value ranges through the use of the optional To keyword (numeric values only). When using the To keyword, the range must be in ascending order (lower to higher). If none of the Case values are true, then the Default code will be executed (if specified).
Note: Select will accept floats as <expression1> but will round them down to the nearest integer (comparisons will be done only with integer values).

Example: Simple example

```
1   Value = 2
2
3   Select Value
4     Case 1
5       Debug "Value = 1"
6
7     Case 2
8       Debug "Value = 2"
9
10    Case 20
11      Debug "Value = 20"
12
13    Default
14      Debug "I don't know"
15  EndSelect
```

Example: Multicase and range example

```
Value = 2

Select Value
  Case 1, 2, 3
    Debug "Value is 1, 2 or 3"

  Case 10 To 20, 30, 40 To 50
    Debug "Value is between 10 and 20, equal to 30 or between 40 and 50"

  Default
    Debug "I don't know"

EndSelect
```

Chapter 66

Using several PureBasic versions on Windows

Overview

It is possible to install several PureBasic versions on your hard disk at the same time. This is useful to finish one project with an older PureBasic version, and start developing a new project with a new PureBasic version.

How to do it

Create different folders like "PureBasic_v3.94" and "PureBasic_v4" and install the related PureBasic version in each folders.
When one "PureBasic.exe" is started, it will assign all ".pb" files with this version of PureBasic. So when a source code is loaded by double-clicking on the related file, the currently assigned PureBasic version will be started. Beside PureBasic will change nothing, which can affect other PureBasic versions in different folders.
To avoid the automatic assignment of ".pb" files when starting the IDE, a shortcut can be created from PureBasic.exe with "/NOEXT" as parameter. The command line options for the IDE are described here .
Note: Since PureBasic 4.10, the settings for the IDE are no longer saved in the PureBasic directory but rather in the %APPDATA%\PureBasic directory. To keep the multiple versions from using the same configuration files, the /P /T and /A switches can be used. Furthermore there is the /PORTABLE switch which puts all files back into the PureBasic directory and disabled the creation of the .pb extension.

Chapter 67

Shared

Syntax

```
Shared <variable> [, ...]
```

Description

Shared allows a variable , an array , a list or a map to be accessed within a procedure . When Shared is used with an array, a list or a map, only the name followed by '()' must be specified.

Example: With variable

```
a = 10

Procedure Change()
  Shared a
  a = 20
EndProcedure

Change()
Debug a   ; Will print 20, as the variable has been shared.
```

Example: With array and list

```
Dim Array(2)
NewList List()
AddElement(List())

Procedure Change()
  Shared Array(), List()
  Array(0) = 1
  List() = 2
EndProcedure

Change()
Debug Array(0)  ; Will print 1, as the array has been shared.
Debug List()    ; Will print 2, as the list has been shared.
```

Chapter 68

Static

Syntax

```
Static[.<type>] <variable[.<type>]> [= <constant expression>] [, ...]
```

Description

Static allows to create a local persistent variable in a Procedure even if the same variable has been declared as Global in the main program. If a type is specified after Static, the default type is changed for this declaration. Static can also be used with arrays , lists and maps . When declaring a static array, the dimension parameter has to be a constant value.
The value of the variable isn't reinitialized at each procedure call, means you can use local variables parallel to global variables (with the same name), and both will keep their values. Each variable can have a default value directly assigned to it, but it has to be a constant value.
Beside Static you can use the keyword Protected , to separate global from local variables, but with Protected the local variables will not keep their values.

Example: With variable

```
1  Global a
2  a = 10
3
4  Procedure Change()
5    Static a
6    a+1
7    Debug "In Procedure: "+Str(a) ; Will print 1, 2, 3 as the variable
       increments at each procedure call.
8  EndProcedure
9
10 Change()
11 Change()
12 Change()
13 Debug a ; Will print 10, as the static variable doesn't affect global
     one.
```

Example: With array

```
1  Global Dim Array(2)
2  Array(0) = 10
```

```
3
4   Procedure Change()
5     Static Dim Array(2)
6     Array(0)+1
7     Debug "In Procedure: "+Str(Array(0)) ; Will print 1, 2, 3 as the
      value of the array field increments at each procedure call.
8   EndProcedure
9
10  Change()
11  Change()
12  Change()
13  Debug Array(0) ; Will print 10, as the static array doesn't affect
      global one.
```

Example: With multiple procedure

```
1   Procedure Foo()
2     Static x = 100  ; the declaration and the assignment is done only
      once at program start.
3
4     Debug x
5     x + 1
6   EndProcedure
7
8   Foo() ; Display 100
9   Foo() ; Display 101
10  Foo() ; Display 102
11
12  Debug "---"
13
14  Procedure Bar()
15    Static x  ; the declaration is done only once at program start.
16    x = 100   ; the assignment is done on every Procedure call.
17
18    Debug x
19    x + 1
20  EndProcedure
21
22  Bar() ; Display 100
23  Bar() ; Display 100
24  Bar() ; Display 100
```

Chapter 69

Structures

Syntax

```
Structure <name> [Extends <name>] [Align <numeric constant expression>]
  ...
EndStructure
```

Description

Structure is useful to define user type, and access some OS memory areas. Structures can be used to enable faster and easier handling of data files. It is very useful as you can group into the same object the information which are common. Structures fields are accessed with the \ option. Structures can be nested. Static arrays are supported inside structures.
Structure fields must have an explicit type among all Basic Types supported by PureBasic, i.e. Byte (.b), Ascii (.a), Character (.c), Word (.w) , Unicode (.u), Long (.l), Integer (.i), Float (.f), Quad (.q), Double (.d)), String (.s) and Fixed String (.s{ Length}).
Dynamic objects like arrays, lists and maps are also supported inside structure and are automatically initialized when the object using the structure is created. To declare such field, use the following keywords: Array, List and Map.
Generally, structures are used in association with a variable, an array , a list , or a map . However, advanced users will be able to allocate a memory structure with AllocateStructure() and free it with FreeStructure() . It is also possible to initialize a structure in memory with InitializeStructure() , to copy it with CopyStructure() , reset it to zero with ClearStructure() and reinitialize it with ResetStructure()
It's possible to perform a full structure copy by using the equal affectation between two structure element of the same type.
The optional Extends parameter allows to extends another structure with new fields. All fields found in the extended structure will be available in the new structure and will be placed before the new fields. This is useful to do basic inheritance of structures.
For advanced users only. The optional Align parameter allows to adjust alignment between every structure field. The default alignment is 1, meaning no alignment. For example, if the alignment is set to 4, every field offset will be on a 4 byte boundary. It can help to get more performance while accessing structure fields, but it can use more memory, as some space between each fields will be wasted. The special value #PB_Structure_AlignC can be used to align the structure as it would be done in language C, useful when importing C structures to use with API functions.
SizeOf can be used with structures to get the size of the structure and OffsetOf can be used to retrieve the address offset of the specified field.
Please note, that in structures a **static array[]** doesn't behave like the normal BASIC array (defined using Dim) to be conform to the C/C++ structure format (to allow direct API structure porting). This means that a**[**2**]** will allocate an array from 0 to 1 where Dim a**(**2**)** will allocate an array from 0 to 2. And library functions Array don't work with them.
When using pointers in structures, the '*' has to be omitted when using the field, once more to ease API code porting. It can be seen as an oddity (and to be honest, it is) but it's like that since the very start of

PureBasic and many, many sources rely on that so it won't be changed.
When using a lot of structure fields you can use the With : EndWith keywords to reduce the amount of code to type and ease its readability.

Example

```
Structure Person
  Name.s
  ForName.s
  Age.w
EndStructure

Dim MyFriends.Person(100)

; Here the position '0' of the array MyFriend()
; will contain one person and it's own information

MyFriends(0)\Name = "Andersson"
MyFriends(0)\Forname = "Richard"
MyFriends(0)\Age = 32
```

Example: A more complex structure (Nested and static array)

```
Structure Window
  *NextWindow.Window  ; Points to another window object
  x.w
  y.w
  Name.s[10]  ; 10 Names available (from 0 to 9)
EndStructure
```

Example: Extended structure

```
Structure MyPoint
  x.l
  y.l
EndStructure

Structure MyColoredPoint Extends MyPoint
  color.l
EndStructure

ColoredPoint.MyColoredPoint\x = 10
ColoredPoint.MyColoredPoint\y = 20
ColoredPoint.MyColoredPoint\color = RGB(255, 0, 0)
```

Example: Structure copy

```
Structure MyPoint
  x.l
  y.l
EndStructure

```

```
  LeftPoint.MyPoint\x = 10
  LeftPoint\y = 20

  RightPoint.MyPoint = LeftPoint

  Debug RightPoint\x
  Debug RightPoint\y
```

Example: Dynamic object

```
  Structure Person
    Name$
    Age.l
    List Friends$()
  EndStructure

  John.Person
  John\Name$ = "John"
  John\Age   = 23

  ; Now, add some friends to John
  ;
  AddElement(John\Friends$())
  John\Friends$() = "Jim"

  AddElement(John\Friends$())
  John\Friends$() = "Monica"

  ForEach John\Friends$()
    Debug John\Friends$()
  Next
```

Example: Static, dynamic array and passing a structure to a procedure

```
  Structure Whatever
    a.l
    b.l[2]          ; Static array (Standard C) with 2 values b[0] and b[1], not resizable
    Array c.l(3,3)  ; Dynamic array with 16 values c(0,0) to c(3,3), resizable with ReDim()
  EndStructure

  MyVar.Whatever

  Procedure MyProcedure(*blahblah.Whatever)
    *blahblah\a = 5
    *blahblah\b[0] = 1
    *blahblah\b[1] = 2
    *blahblah\c(3,3) = 33
  EndProcedure

  MyProcedure(@MyVar)
  Debug MyVar\a
  Debug MyVar\b[0]
  Debug MyVar\b[1]
```

```
Debug MyVar\c(3,3)

;Debug MyVar\c(0,10) ; Out-of-bounds index error
ReDim MyVar\c(3,10)  ; Beware, only the last dimension can be resized!
Debug  MyVar\c(0,10) ; Cool, the array is bigger now!
```

Example: Nested structure(s)

```
Structure pointF
  x.f
  y.f
EndStructure

Structure Field
  Field1.q
  Field2.s{6}
  Field3.s
  Array Tab.pointF(3)
EndStructure

Define MyVar.Field

MyVar\Tab(3)\x = 34.67
```

Example: Alignment

```
Structure Type Align 4
  Byte.b
  Word.w
  Long.l
  Float.f
EndStructure

Debug OffsetOf(Type\Byte)   ; will print 0
Debug OffsetOf(Type\Word)   ; will print 4
Debug OffsetOf(Type\Long)   ; will print 8
Debug OffsetOf(Type\Float)  ; will print 12
```

Example: Pointers

```
Structure Person
  *Next.Person ; Here the '*' is mandatory to declare a pointer
  Name$
  Age.b
EndStructure

Timo.Person\Name$ = "Timo"
Timo\Age = 25

Fred.Person\Name$ = "Fred"
Fred\Age = 25

Timo\Next = @Fred ; When using the pointer, the '*' is omitted
```

```
14
15  Debug Timo\Next\Name$ ; Will print 'Fred'
```

Syntax

```
StructureUnion
  Field1.Type
  Field2.Type
  ...
EndStructureUnion
```

Description

Structure union are only useful for advanced programmers who want to save some memory by sharing some fields inside the same structure. It's like the 'union' keyword in C/C++.
Note: Each field in the StructureUnion declaration can be of a different type .

Example

```
1  Structure Type
2    Name$
3    StructureUnion
4      Long.l      ; Each field (Long, Float and Byte) resides at the
     same address in memory.
5      Float.f     ;
6      Byte.b      ;
7    EndStructureUnion
8  EndStructure
```

Example: Example RGB

```
1  Structure StrColor
2    Red.a
3    Green.a
4    Blue.a
5    Alpha.a
6  EndStructure
7
8  Structure StrColorU
9    StructureUnion
10     Component.StrColor
11     Color.l
12     Byte.a[4]
13   EndStructureUnion
14 EndStructure
15
16 Define Color1.StrColorU
17
18 Color1\Color = RGBA($10,$20,$30,$FF)
19 Debug "hex = " + Hex(Color1\Color, #PB_Long)  ;hex = FF302010
     (little-endian cpu)
20 Debug "r = " + Hex(Color1\Component\Red)      ;r = 10
21 Debug "g = " + Hex(Color1\Component\Green)    ;g = 20
```

```
Debug "b = " + Hex(Color1\Component\Blue)      ;b = 30
Debug "a = " + Hex(Color1\Component\Alpha)     ;a = FF

Debug "by0 = " + Hex(Color1\Byte[0])           ;by0 = 10
Debug "by1 = " + Hex(Color1\Byte[1])           ;by1 = 20
Debug "by2 = " + Hex(Color1\Byte[2])           ;by2 = 30
Debug "by3 = " + Hex(Color1\Byte[3])           ;by3 = FF
```

Example: Example date handling

```
Structure date ; jj.mm.yyyy
  day.s{2}
  dot.s{1}
  month.s{2}
  dot.s{1}
  year.s{4}
EndStructure

Structure dateU
  StructureUnion
    s.s{10}
    d.date
  EndStructureUnion
EndStructure

Dim d1.dateU(365)

;An array
d1(0)\s = "05.04.2028"

Debug d1(0)\d\day
Debug d1(0)\d\month
Debug d1(0)\d\year

;A variable
d2.date2\s = "15.11.2030"

Debug d2\d\day
Debug d2\d\month
Debug d2\d\year
```

Chapter 70

Subsystems

Introduction

PureBasic integrated commandset relies on available OS libraries. Sometimes, there is different way to achieve the same goal and when it makes sense, PureBasic offers the possibility to change the used underlying libraries for specific commands, without changing one line of source code. For example, on Windows there is the 'DirectX11' subsystem available, which will use DirectX functions to render sprites, instead of OpenGL (which is the default subsystem). It can be useful to use DirectX instead of OpenGL when wanting a bit quicker rendering on Windows.
To enable a subsystem, its name has to be set in the IDE compiler options , or through the /SUBSYSTEM command-line switch. The subsystem name is not case-sensitive. This is a compile time option, which means an executable can not embed more than one subsystem at once for a specific library. If multiple support is needed (for example shipping an OpenGL and DirectX version of a game), two executables needs to be created.
The available subsystems are located in the PureBasic 'subsystems' folder. Any residents or libraries found in this drawer will have precedency over the default libraries and residents, when a subsystem is specified. Any number of different subsystems can be specified (as long it doesn't affect the same libraries).
The Subsystem compiler function can be used to detect if a specific subsystem is used for the compilation.

Available subsystems

Here is a list of available subsystems, and the affected libraries:
Windows

```
DirectX9: use DirectX 9 instead of OpenGL. Affected libraries:
        - Sprite
        - Screen
        - Note: 3D engine no more available as it uses OpenGL
DirectX11: use DirectX 11 instead of OpenGL. Affected libraries:
        - Sprite
        - Screen
        - Note: 3D engine no more available as it uses OpenGL
```

Linux

```
Gtk2: Use GTK2 instead of GTK3 for GUI toolkit. Affected libraries:
        - 2D Drawing
        - AudioCD
        - Clipboard
        - Desktop
```

```
        - Drag & Drop
        - Font
        - Gadget
        - Image
        - Menu
        - Movie
        - Printer
        - Requester
        - Scintilla
        - StatusBar
        - SysTray
        - Toolbar
        - Window

  Qt: Use QT instead of GTK3 for GUI toolkit. Affected libraries:
        - 2D Drawing
        - AudioCD
        - Clipboard
        - Desktop
        - Drag & Drop
        - Font
        - Gadget
        - Image
        - Menu
        - Movie
        - Printer
        - Requester
        - Scintilla
        - StatusBar
        - SysTray
        - Toolbar
        - Window
```

MacOS X

```
  None
```

Chapter 71

Threaded

Syntax

```
Threaded[.<type>] <variable[.<type>]> [= <constant expression>] [, ...]
```

Description

Threaded allows to create a thread based persistent variable , arrays (except multi-dimensional arrays), lists or maps . This means every thread will have its own version of the object. This is only useful when writing multithreaded programs. If a type is specified after Threaded, the default type is changed for this declaration.
Each variable can have a default value directly assigned to it, but it has to be a constant value.
Threaded initialization is done at thread first start. That implies when declaring and assigning a threaded variable in the same time, the variable is assigned for all threads. See example 2. When declaring a threaded array, the dimension parameter has to be a constant value.
A Threaded object can't be declared in a procedure, its scope is always global.

Example: 1 With variables

```
1  Threaded Counter
2
3  Counter = 128
4
5  Procedure Thread(Parameter)
6
7    Debug Counter ; Will display zero as this thread doesn't have used
     this variable for now
8    Counter = 256
9    Debug Counter ; Will display 256
10
11 EndProcedure
12
13 Thread = CreateThread(@Thread(), 0)
14 WaitThread(Thread) ; Wait for thread ending
15
16 Debug Counter ; Will display 128, even if Counter has been changed in
    the thread
```

Example: 2 All threads

```
1   Threaded Counter = 128 ; Defined for all threads
2
3   Procedure Thread(Parameter)
4
5     Debug Counter ; Will display 128 because a programme started,
      starts a thread too
6     Counter = 256
7     Debug Counter ; Will display 256
8
9   EndProcedure
10
11  Thread = CreateThread(@Thread(), 0)
12  WaitThread(Thread) ; Wait for thread ending
13
14  Debug Counter ; Will display 128, even if Counter has been changed in
     the thread
```

Chapter 72

UserGuide - Advanced functions

~~<empty for now>~~

UserGuide Navigation

< Previous: Other Library functions || Overview || Next: Some Tips & Tricks >

Chapter 73

UserGuide - Constants

In addition to variables PureBasic provides a method to define constants too. In fact it provides several. Well have a quick look at them now. Predefined constants - provided either by PureBasic itself (all begin with #PB_), or from the API for the operating system. The IDEs Structure Viewer tool has a panel which shows all the predefined constants. User defined constants - by defining a constant name with the prefix # you can provide your own constants to make code more readable.

```
#MyConstant1 = 10
#MyConstant2 = "Hello, World!"
```

Enumerations PureBasic will automatically number a series of constants sequentially in an Enumeration, by default enumerations will begin from zero but this can be altered, if desired.

```
Enumeration
?? #MyConstantA
?? #MyConstantB
?? #MyConstantC
EndEnumeration

Enumeration 10 Step 5
?? #MyConstantD
?? #MyConstantE
?? #MyConstantF
EndEnumeration

Debug #MyConstantA    ; will be 0
Debug #MyConstantB    ; will be 1
Debug #MyConstantC    ; will be 2

Debug #MyConstantD    ; will be 10
Debug #MyConstantE    ; will be 15
Debug #MyConstantF    ; will be 20
```

UserGuide Navigation

< Previous: Variables || Overview || Next: Decisions & Conditions >

Chapter 74

UserGuide - Storing data in memory

This example gathers information about the files in the logged on user's home directory into a structured list . For now the output isn't very exciting but we will come back to this example later on and make it a bit more friendly in several different ways.

```
1   ; This section describes the fields of a structure or record, mostly
     integers in this case,
2   ; but notice the string for the file name and the quad for the file
     size.
3   Structure FILEITEM
4     Name.s
5     Attributes.i
6     Size.q
7     DateCreated.i
8     DateAccessed.i
9     DateModified.i
10  EndStructure
11
12  ; Now we define a new list of files using the structure previously
     specified
13  ; and some other working variables we'll use later on.
14  NewList Files.FILEITEM()
15  Define.s Folder
16  Define.l Result
17
18  ; This function gets the home directory for the logged on user.
19  Folder = GetHomeDirectory()
20
21  ; Open the directory to enumerate all its contents.
22  Result = ExamineDirectory(0, Folder, "*.*")
23
24  ; If this is ok, begin enumeration of entries.
25  If Result
26    ; Loop through until NextDirectoryEntry(0) becomes zero -
     indicating that there are no more entries.
27    While NextDirectoryEntry(0)
28      ; If the directory entry is a file, not a folder.
29      If DirectoryEntryType(0) = #PB_DirectoryEntry_File
30        ; Add a new element to the list.
31        AddElement(Files())
32        ; And populate it with the properties of the file.
33        Files()\Name = DirectoryEntryName(0)
34        Files()\Size = DirectoryEntrySize(0)
```

```
        Files()\Attributes = DirectoryEntryAttributes(0)
        Files()\DateCreated = DirectoryEntryDate(0, #PB_Date_Created)
        Files()\DateAccessed = DirectoryEntryDate(0, #PB_Date_Accessed)
        Files()\DateModified = DirectoryEntryDate(0, #PB_Date_Modified)
      EndIf
    Wend
    ; Close the directory.
    FinishDirectory(0)
  EndIf

  ; Shows the results in the debug window (if there is no entry,
   nothing will be displayed)
  ForEach Files()
    Debug "Filename = " + Files()\Name
    Debug "Size = " + Str(Files()\Size)
    Debug "Attributes = " + StrU(Files()\Attributes)
    Debug "Created = " + StrU(Files()\DateCreated)
    Debug "Accessed = " + StrU(Files()\DateAccessed)
    Debug "Modified = " + StrU(Files()\DateModified)
  Next Files()
```

Ok, firstly, the dates in the output are just numbers - this isn't very helpful, so let's make them look a bit more familiar. Replace the last three Debug statements with these:

```
      ...
      Debug "Created = " + FormatDate("%dd/%mm/%yyyy",
   Files()\DateCreated)
      Debug "Accessed = " + FormatDate("%dd/%mm/%yyyy",
   Files()\DateAccessed)
      Debug "Modified = " + FormatDate("%dd/%mm/%yyyy",
   Files()\DateModified)
```

The FormatDate() function takes a date in PureBasic's own numeric date format and displays it in a format that we can specify. So now things should begin to look a bit more sensible.
Finally, for now, the list isn't in any particular order, so let's sort the list before we display it. Add this line before the comment about showing the list and the ForEach loop.

```
  ...

  ; Sort the list into ascending alphabetical order of file name.
  SortStructuredList(Files(), #PB_Sort_Ascending,
   OffsetOf(FILEITEM\Name), #PB_String)

  ; If there are some entries in the list, show the results in the
   debug window.
  ...
```

This command takes the structured list , and resorts it into ascending order (#PB_Sort_Ascending), of the Name field of the structure (OffsetOf(FILEITEM\Name)), which is a string value (#PB_String).

UserGuide Navigation

< Previous: String Manipulation || Overview || Next: Input & Output >

Chapter 75

UserGuide - Decisions & Conditions

There are different ways of processing data obtained from user input or other way (loading from a file, ...). The common arithmetic functions (+, -, *, /, ...) can be combined with conditions. You can use the If : Else/ElseIf : EndIf set of keywords or the Select : Case/Default : EndSelect keywords, just use what is the best for your situation!
This example shows the use of If : ElseIf : Else : EndIf for creating a message, possibly for showing in the status bar of a form (GUI) or something similar, based upon the number of items and filtered items in an, imaginary, list. Note that unlike some other BASIC languages, PureBasic doesn't use the "Then" keyword and that there is no space in the ElseIf and EndIf keywords.

```
1   Define.l Items = 10, Filter = 6
2   Define.s Message
3
4   If Items = 0
5     Message = "List is empty."
6
7   ElseIf Items = 1 And Filter = 0
8     Message = "One item. Not shown by filter."
9
10  ElseIf Items > 1 And Filter = 0
11    Message = StrU(Items) + " items. All filtered."
12
13  ElseIf Items > 1 And Filter = 1
14    Message = StrU(Items) + " items. One shown by filter."
15
16  ElseIf Items = Filter
17    Message = StrU(Items) + " items. None filtered."
18
19  Else
20    ; None of the other conditions were met.
21    Message = StrU(Items) + " items. " + StrU(Filter) +" shown by
     filter."
22
23  EndIf
24
25  Debug Message
```

This example shows the use of Select : Case : Default : EndSelect to categorize the first 127 ASCII characters into groups. The "For : Next" loop counts to 130 to demonstrate the Default keyword.

```
1   Define.c Char
2   Define.s Message
3
```

```
For Char = 0 To 130

  Select Char

    Case 0 To 8, 10 To 31, 127
      Message = StrU(Char) + " is a non-printing control code."

    Case 9
      Message = StrU(Char) + " is a tab."

    Case 32
      Message = StrU(Char) + " is a space."

    Case 36, 128
      Message = StrU(Char) + " is a currency symbol. (" + Chr(Char) + ")"

    Case 33 To 35, 37 To 47, 58 To 64, 91 To 96
      Message = StrU(Char) + " is a punctuation mark or math symbol. (" + Chr(Char) + ")"

    Case 48 To 57
      Message = StrU(Char) + " is a numeral. (" + Chr(Char) + ")"

    Case 65 To 90
      Message = StrU(Char) + " is an upper case letter. (" + Chr(Char) + ")"

    Case 97 To 122
      Message = StrU(Char) + " is a lower case letter. (" + Chr(Char) + ")"

    Default
      ; If none of the preceding Case conditions are met.
      Message = "Sorry, I don't know what " + StrU(Char) + " is!"

  EndSelect

  Debug Message

Next Char
```

UserGuide Navigation

< Previous: Constants || Overview || Next: Loops >

Chapter 76

UserGuide - Compiler directives (for different behavior on different OS)

This will be our last visit to the File Properties program. There is one limitation discussed previously to overcome and we've left it until now because it is a special case.
So far the Attributes column on the display has simply been an integer . This is because the return values of the GetFileAttributes() and DirectoryEntryAttributes() instructions have a different meaning on Windows systems compared with Mac and Linux systems.
We can't allow for this difference at run-time, however we can use Compiler Directives to have the program behave differently on the three different operating systems.
Add this new procedure declaration to that section.

```
1  Declare.s AttributeString(Attributes.l)
```

Add this new procedure to the implementation section.

```
1  Procedure.s AttributeString(Attributes.l)
2    ; Converts an integer attributes value into a string description.
3    ; Supports Linux, Mac and Windows system's attributes.
4
5    Protected.s Result
6
7    Result = ""
8
9    CompilerIf #PB_Compiler_OS = #PB_OS_Windows
10     ?? ??
11     ?? ?? ; These are the attributes for Windows systems.
12     ?? ?? ; A logical-and of the attribute with each constant filters
   out that bit and can then be used for comparison.
13
14     ?? ?? If Attributes & #PB_FileSystem_Archive
15       ?? ?? ?? Result + "A"
16     ?? ?? Else
17       ?? ?? ?? Result + " "
18     ?? ?? EndIf
19     ?? ??
20     ?? ?? If Attributes & #PB_FileSystem_Compressed
21       ?? ?? ?? Result + "C"
22     ?? ?? Else
23       ?? ?? ?? Result + " "
24     ?? ?? EndIf
25     ?? ??
26     ?? ?? If Attributes & #PB_FileSystem_Hidden
```

```
      ?? ?? ?? Result + "H"
    ?? ?? Else
      ?? ?? ?? Result + " "
    ?? ?? EndIf
    ?? ??
    ?? ?? If Attributes & #PB_FileSystem_ReadOnly
      ?? ?? ?? Result + "R"
    ?? ?? Else
      ?? ?? ?? Result + " "
    ?? ?? EndIf
    ?? ??
    ?? ?? If Attributes & #PB_FileSystem_System
      ?? ?? ?? Result + "S"
    ?? ?? Else
      ?? ?? ?? Result + " "
    ?? ?? EndIf
    ?? ??
  CompilerElse

    ; These are the attributes for Mac and Linux systems.

    If Attributes & #PB_FileSystem_Link
      Result + "L "
    Else
      Result + "  "
    EndIf

    ; User attributes.
    If Attributes & #PB_FileSystem_ReadUser
      Result + "R"
    Else
      Result + " "
    EndIf

    If Attributes & #PB_FileSystem_WriteUser
      Result + "W"
    Else
      Result + " "
    EndIf

    If Attributes & #PB_FileSystem_ExecUser
      Result + "X "
    Else
      Result + "  "
    EndIf

    ; Group attributes.
    If Attributes & #PB_FileSystem_ReadGroup
      Result + "R"
    Else
      Result + " "
    EndIf

    If Attributes & #PB_FileSystem_WriteGroup
      Result + "W"
    Else
      Result + " "
    EndIf

```

```
86        If Attributes & #PB_FileSystem_ExecGroup
87          Result + "X "
88        Else
89          Result + "  "
90        EndIf
91
92        ; All attributes.
93        If Attributes & #PB_FileSystem_ReadAll
94          Result + "R"
95        Else
96          Result + " "
97        EndIf
98
99        If Attributes & #PB_FileSystem_WriteAll
100         Result + "W"
101       Else
102         Result + " "
103       EndIf
104
105       If Attributes & #PB_FileSystem_ExecAll
106         Result + "X"
107       Else
108         Result + " "
109       EndIf
110
111     CompilerEndIf
112
113     ; Return the result.
114     ProcedureReturn Result
115
116   EndProcedure
```

Finally, replace these two lines in the ListLoad procedure

```
1       ; Convert the attributes to a string, for now.
2       Attrib = StrU(Files()\Attributes)
```

with these,

```
1       ; Call AttributeString to convert the attributes to a string
    representation.
2       Attrib = AttributeString(Files()\Attributes)
```

Now when the program is executed a string display will be shown instead of the integer values. On a Windows system it would look something like this (assuming all attributes are set):

```
1   ACHRS
```

and on the other two systems:

```
1   L RWX RWX RWX
```

The CompilerIf instruction works much like an If instruction - however it is the compiler that makes the decision at compile-time, rather than the executable at run-time. This means that we can include different code to handle the file attributes on the different operating systems.
The lines between:

```
1   CompilerIf #PB_Compiler_OS = #PB_OS_Windows
```

and

```
  CompilerElse
```

will be compiled on Windows systems. The constant #PB_Compiler_OS is automatically defined by PureBasic to allow this kind of program logic.
The other section will be used on Mac and Linux systems - which work the same way, conveniently. If these operating systems had different attribute values too, then we could use CompilerSelect and CompilerCase to make a three-way determination.

```
  CompilerSelect #PB_Compiler_OS

    CompilerCase #PB_OS_Linux
      ; Code for Linux systems.


    CompilerCase #PB_OS_MacOS
      ; Code for Mac systems.

    CompilerCase #PB_OS_Windows
      ; Code for Windows systems.

  CompilerEndSelect
```

The last compiler directive that we're going to discuss here is: EnableExplicit .
There is a good reason for using this directive. It requires that all variables must be defined explicitly before usage, in some way, (using Define , Dim , Global , Protected , Static etc.) Doing so eliminates the possibility of logic errors due to mistyped variable names being defined "on-the-fly". This type of subtle error will not affect a program's compilation but could well present as an inconvenient bug at run-time. Using this directive eliminates this kind of problem as a compiler error would occur.
For example:

```
  EnableExplicit

  Define.l Field, FieldMax

  ; ...

  If Field < FieldMax
    ; If EnableExplicit is omitted this section of code may not execute
   when intended because FieldMax will be zero.
  EndIf
```

UserGuide Navigation

< Previous: Structuring code in Procedures || Overview || Next: Reading and writing files >

Chapter 77

UserGuide - Reading and writing files

This example will write 100 random records each containing a byte , a floating-point number , a long integer and a string . It then reads all the records back and displays them in the debug window .
It demonstrates the GetTemporaryDirectory() , CreateFile() , OpenFile() , Eof() and a number of Read and Write data instructions too.
It works fine as far as it goes, but has a drawback. As the string value has a variable length - you can't randomly access the records because you can't predict where each new record will start in the file. They must be all be read back in the same sequence as they were written. This isn't a problem with the small number of records created here but this could be an inconvenience with a larger file. PureBasic offers a way to handle this situation too - but an example would be too complex for this topic. See the Database sections of the help-file or reference manual to see how it could be done.

```
1   EnableExplicit
2   ; Define the constants:
3   #MAXBYTE = 255
4   #MAXLONG = 2147483647
5
6   ; Define some variables.
7   Define.f Float
8   Define.i Count, File
9   Define.s Folder, FileName, String
10
11  ; Create a temporary file name.
12  Folder = GetTemporaryDirectory()
13  FileName = Folder + "test.data"
14
15  ; Create the temporary file.
16  ; If #PB_Any is used, CreateFile returns the file's number.
17  ; Useful if you may have more than one file open simultaneously.
18  File = CreateFile(#PB_Any, FileName)
19
20  If File
21    ; If this was successful - write 100 random records.
22    For Count = 1 To 100
23
24      ; Write a random byte (0 - 255).
25      WriteByte(File, Random(#MAXBYTE))
26
27      ; Create and write a random float.
28      ; This calculation is there to make the number have a
    floating-point component (probably).
29      Float = Random(#MAXLONG) / ((Random(7) + 2) * 5000)
30      WriteFloat(File, Float)
```

```
31
32          ; Write a random long.
33          WriteLong(File, Random(#MAXLONG))
34
35          ; Create and write a random string in Unicode format.
36          ; Note the use of WriteStringN to delimit the string with an end
        of line marker.
37          String = "String " + StrU(Random(#MAXLONG))
38          WriteStringN(File, String, #PB_Unicode)
39
40        Next Count
41
42        ; Close the file.
43        CloseFile(File)
44
45      Else
46        ; If this was unsuccessful.
47        Debug "Could not create the file: " + FileName
48
49      EndIf
50
51      ; Open the file for reading this time.
52      File = ReadFile(#PB_Any, FileName)
53
54      If File
55        ; If this was successful - read and display records from the file.
56
57        ; Reset the counter.
58        Count = 1
59
60        ; Loop until the 'end of file' is reached.
61        ; This will read all of the records regardless of how many there
        are.
62        While Eof(File) = 0
63
64          ; Print a header line.
65          Debug
        "----------------------------------------------------------------------------------------------------"
66
67          Debug "[" + StrU(Count) + "]"
68          Count + 1
69          ; Read a byte value and print it.
70          Debug StrU(ReadByte(File), #PB_Byte)
71
72          ; Read a float value.
73          Debug StrF(ReadFloat(File))
74
75          ; Read a long value.
76          Debug StrU(ReadLong(File), #PB_Long)
77
78          ; Read a string value.
79          Debug ReadString(File, #PB_Unicode)
80
81        Wend
82
83        ; Print the trailing line.
84        Debug
        "----------------------------------------------------------------------------------------------------"
85
```

```
86      ; Close the file.
87      CloseFile(File)
88
89      ; Tidy up.
90      DeleteFile(FileName)
91
92    Else
93      ; If this was unsuccessful.
94      Debug "Could not open the file: " + FileName
95
96    EndIf
```

UserGuide Navigation

< Previous: Compiler directives || Overview || Next: Memory access >

Chapter 78

UserGuide - First steps with the Debug output (variables & operators)

Normally we would present here the typical "Hello World". You want to see it? Ok here we go with two examples:

Debug output

"Hello World" in the Debug output window:

```
1  Debug "Hello World!"
```

MessageRequester

"Hello World" in a MessageRequester() :

```
1  MessageRequester("", "Hello World!")
```

Advanced Debug example

We now continue with a short example using the available variable types, arithmetic operators and displaying the result:

```
1  a = 5
2  b = 7
3  c = 3
4
5  d = a + b  ; we use the values stored in variables 'a' and 'b' and
     save the sum of them in 'd'
6  d / c      ; we directly use the value of 'd' (= 12) divided by the
     value of 'c' (= 3) and save the result in 'd'
7
8  Debug d    ; will output 4
```

This way you have used variables "on the fly". To force the compiler always declaring variables before their first use, just use the keyword EnableExplicit .

UserGuide Navigation

Overview || Next: Variables >

Chapter 79

UserGuide - Displaying graphics output & simple drawing

This example show how to create a simple drawing. It uses the 2D drawing commands to draw two sine waves at different frequencies and shows the harmonic produced by combining the two waves. It uses procedures , which we will discuss in more detail later on, to break the drawing tasks into three self-contained tasks:
Drawing the axes - demonstrates the Line() command.
Drawing the legend - demonstrates the Box() and DrawText() commands.
Drawing the wave forms - demonstrates the LineXY() command and shows how to use color.

```
1   ; Window
2   Enumeration
3     #WinHarmonic
4   EndEnumeration
5
6   ; Gadgets
7   Enumeration
8     #txtPlot1
9     #cboPlot1
10    #txtPlot2
11    #cboPlot2
12    #imgPlot
13  EndEnumeration
14
15  ; Image
16  Enumeration
17    #drgPlot
18  EndEnumeration
19
20  ; Image dimensions are used in several places so define constants.
21  #imgPlotX = 8
22  #imgPlotY = 40
23  #imgPlotW = 745
24  #imgPlotH = 645
25
26  ; Event variables
27  Define.l Event, EventWindow, EventGadget, EventType, EventMenu
28
29  ; Implementation
30  Procedure CreateWindow()
31    ; Creates the window and gadgets.
32
```

```
33      If OpenWindow(#WinHarmonic, 30, 30, #imgPlotW + 20, #imgPlotH + 55,
        "Harmonics", #PB_Window_SystemMenu | #PB_Window_MinimizeGadget |
        #PB_Window_TitleBar)
34
35        ; This is a non-visual gadget used to draw the image, later its
        contents will be displayed in #imgPlot.
36        CreateImage(#drgPlot, #imgPlotW - 5, #imgPlotH - 5, 24)
37
38        ; Label for the Plot 1 combo.
39        TextGadget(#txtPlot1, 2, 5, 50, 25, "Plot 1:")
40
41        ; The Plot 1 combo.
42        ComboBoxGadget(#cboPlot1, 55, 5, 150, 25)
43        AddGadgetItem(#cboPlot1, 0, "Sin(X)")
44        AddGadgetItem(#cboPlot1, 1, "Sin(X * 2)")
45        AddGadgetItem(#cboPlot1, 2, "Sin(X * 3)")
46        AddGadgetItem(#cboPlot1, 3, "Sin(X * 4)")
47        AddGadgetItem(#cboPlot1, 4, "Sin(X * 5)")
48        AddGadgetItem(#cboPlot1, 5, "Sin(X * 6)")
49
50        ; Select Sin(X)
51        SetGadgetState(#cboPlot1, 0)
52
53        ; Label for the Plot 2 combo.
54        TextGadget(#txtPlot2, 230, 5, 50, 25, "Plot 2:")
55
56        ; The Plot 2 combo.
57        ComboBoxGadget(#cboPlot2, 280, 5, 150, 25)
58        AddGadgetItem(#cboPlot2, 0, "Sin(X)")
59        AddGadgetItem(#cboPlot2, 1, "Sin(X * 2)")
60        AddGadgetItem(#cboPlot2, 2, "Sin(X * 3)")
61        AddGadgetItem(#cboPlot2, 3, "Sin(X * 4)")
62        AddGadgetItem(#cboPlot2, 4, "Sin(X * 5)")
63        AddGadgetItem(#cboPlot2, 5, "Sin(X * 6)")
64
65        ; Select Sin(X * 2), otherwise the initial display is a bit
        uninteresting.
66        SetGadgetState(#cboPlot2, 1)
67
68        ; The visual image gadget on the window.
69        ImageGadget(#imgPlot, #imgPlotX, #imgPlotY, #imgPlotW, #imgPlotH,
        0, #PB_Image_Border)
70
71      EndIf
72
73  EndProcedure
74
75  Procedure PlotAxes()
76    ; Draws the axes on the image #drgPlot.
77
78    ; Send drawing commands to #drgPlot.
79    StartDrawing(ImageOutput(#drgPlot))
80
81        ; Draw a white background.
82        Box(0, 0, ImageWidth(#drgPlot), ImageHeight(#drgPlot), RGB(255,
        255, 255))
83
84        ; Draw the axes in black.
85        Line(1, 1, 1, ImageHeight(#drgPlot) - 2, RGB(0, 0, 0))
```

```
86        Line(1, (ImageHeight(#drgPlot) - 2) /2, ImageWidth(#drgPlot) -2,
      1, RGB(0, 0, 0))
87
88        ; Finished drawing.
89      StopDrawing()
90    EndProcedure
91
92    Procedure PlotLegend(alngPlot1, alngPlot2)
93      ; Draws the legend on the image #drgPlot.
94
95      Protected.s strFunc1, strFunc2, strLabel1, strLabel2, strLabel3
96
97      ; Set label text 1.
98      If alngPlot1 = 0
99        strFunc1 = "Sin(X)"
100     Else
101       strFunc1 = "Sin(X * " + StrU(alngPlot1 + 1) + ")"
102     EndIf
103
104     ; Set label text 2.
105     If alngPlot2 = 0
106       strFunc2 = "Sin(X)"
107     Else
108       strFunc2 = "Sin(X * " + StrU(alngPlot2 + 1) + ")"
109     EndIf
110
111     ; Set label text.
112     strLabel1 = "Y = " + strFunc1
113     strLabel2 = "Y = " + strFunc2
114     strLabel3 = "Y = " + strFunc1 + " + " + strFunc2
115
116     ; Draw legend.
117     StartDrawing(ImageOutput(#drgPlot))
118
119       ; Box.
120       DrawingMode(#PB_2DDrawing_Outlined)
121       Box(20, 10, TextWidth(strLabel3) + 85, 80, RGB(0, 0, 0))
122
123       ; Label 1.
124       Line(30, 30, 50, 1, RGB(0, 0, 255))
125       DrawText(95, 22, strLabel1, RGB(0, 0, 0), RGB(255, 255, 255))
126
127       ; Label 2.
128       Line(30, 50, 50, 1, RGB(0, 255, 200))
129       DrawText(95, 42, strLabel2, RGB(0, 0, 0), RGB(255, 255, 255))
130
131       ; Label 3.
132       Line(30, 70, 50, 1, RGB(255, 0, 0))
133       DrawText(95, 62, strLabel3, RGB(0, 0, 0), RGB(255, 255, 255))
134
135     StopDrawing()
136
137   EndProcedure
138
139   Procedure PlotFunction(alngPlot1, alngPlot2)
140     ; Draws the waveforms on the image #drgPlot.
141
142     Protected.l lngSX, lngEX
143     Protected.f fltRad1, fltRad2, fltSY1, fltEY1, fltSY2, fltEY2,
```

```
    fltSY3, fltEY3
144
145     StartDrawing(ImageOutput(#drgPlot))
146
147       ; Set initial start points for each wave.
148       lngSX = 1
149       fltSY1 = ImageHeight(#drgPlot) / 2
150       fltSY2 = fltSY1
151       fltSY3 = fltSY1
152
153       ; Plot wave forms.
154       For lngEX = 1 To 720
155         ; Sine function works in radians, so convert from degrees and
    calculate sine.
156
157         ; Function 1
158         If alngPlot1 = 0
159           fltRad1 = Sin(Radian(lngEX))
160         Else
161           ; If the function should have a multiplier, account for this.
162           fltRad1 = Sin(Radian(lngEX) * (alngPlot1 + 1))
163         EndIf
164
165         ; Function 2
166         If alngPlot2 = 0
167           fltRad2 = Sin(Radian(lngEX))
168         Else
169           fltRad2 = Sin(Radian(lngEX) * (alngPlot2 + 1))
170         EndIf
171
172         ; Plot function 1 in blue.
173         ; Calculate end Y point.
174         fltEY1 = (ImageHeight(#drgPlot) / 2) + (fltRad1 * 100)
175         ; Draw a line from the start point to the end point.
176         LineXY(lngSX, fltSY1, lngEX, fltEY1, RGB(0, 0, 255))
177         ; Update the next start Y point to be the current end Y point.
178         fltSY1 = fltEY1
179
180         ; Plot function 2 in green.
181         fltEY2 = (ImageHeight(#drgPlot) / 2) + (fltRad2 * 100)
182         LineXY(lngSX, fltSY2, lngEX, fltEY2, RGB(0, 255, 200))
183         fltSY2 = fltEY2
184
185         ; Plot harmonic in red.
186         fltEY3 = (ImageHeight(#drgPlot) / 2) + ((fltRad1 + fltRad2) *
    100)
187         LineXY(lngSX, fltSY3, lngEX, fltEY3, RGB(255, 0, 0))
188         fltSY3 = fltEY3
189
190         ; Update the start X point to be the current end X point.
191         lngSX = lngEX
192       Next lngEX
193
194     StopDrawing()
195
196   EndProcedure
197
198   ;- Main
199   CreateWindow()
```

```
  PlotAxes()
  PlotLegend(GetGadgetState(#cboPlot1), GetGadgetState(#cboPlot2))
  PlotFunction(GetGadgetState(#cboPlot1), GetGadgetState(#cboPlot2))

  ; Reload the image gadget now drawing is complete.
  ImageGadget(#imgPlot, #imgPlotX, #imgPlotY, #imgPlotW, #imgPlotH,
   ImageID(#drgPlot), #PB_Image_Border)

  ;- Event loop
  Repeat
    Event = WaitWindowEvent()
    EventWindow = EventWindow()
    EventGadget = EventGadget()
    EventType = EventType()

    Select Event
      Case #PB_Event_Gadget
        If EventGadget = #txtPlot1 Or EventGadget = #txtPlot2
          ; Do nothing.
        ElseIf EventGadget = #imgPlot
          ; Do nothing.
        ElseIf EventGadget = #cboPlot1 Or EventGadget = #cboPlot2
          ; If one of the combo boxes changed, redraw the image.
          PlotAxes()
          PlotLegend(GetGadgetState(#cboPlot1),
   GetGadgetState(#cboPlot2))
          PlotFunction(GetGadgetState(#cboPlot1),
   GetGadgetState(#cboPlot2))
          ImageGadget(#imgPlot, #imgPlotX, #imgPlotY, #imgPlotW,
   #imgPlotH, ImageID(#drgPlot), #PB_Image_Border)
        EndIf
      Case #PB_Event_CloseWindow
        If EventWindow = #WinHarmonic
          CloseWindow(#WinHarmonic)
          Break
        EndIf
    EndSelect
  ForEver
```

UserGuide Navigation

< Previous: Building a graphical user interface (GUI) || Overview || Next: Structuring code in Procedures >

Chapter 80

UserGuide - Building a graphical user interface (GUI)

In addition to the console window , PureBasic supports the creation of graphical user interfaces (GUI) too. So let's revisit the file properties example from previous items again and turn it into a GUI application.
Note that PureBasic provides a far easier way of getting this particular job done already - the ExplorerListGadget() ; but, as the example is intended to introduce managing GUI elements, using that gadget would defeat this object a bit.

```
1   ; The structure for file information as before.
2   Structure FILEITEM
3     Name.s
4     Attributes.i
5     Size.q
6     DateCreated.i
7     DateAccessed.i
8     DateModified.i
9   EndStructure
10
11  ; This is a constant to identify the window.
12  Enumeration
13    #WindowFiles
14  EndEnumeration
15
16  ; This is an enumeration to identify controls which will appear on
     the window.
17  Enumeration
18    #Folder
19    #Files
20  EndEnumeration
21
22  ; Now we define a list of files using the structure previously
     specified.
23  NewList Files.FILEITEM()
24
25  ; And some working variables to make things happen.
26  Define.s Access, Attrib, Create, Folder, Modify, Msg, Num, Size
27  Define.l Result, Flags
28
29  ; These variables will receive details of GUI events as they occur in
     the program.
30  Define.l Event, EventWindow, EventGadget, EventType, EventMenu
```

```
31
32    ; This function gets the home directory for the logged on user.
33    Folder = GetHomeDirectory()
34
35    ; Open the directory to enumerate its contents.
36    Result = ExamineDirectory(0, Folder, "*.*")
37
38    ; If this is ok, begin enumeration of entries.
39    If Result
40      ; Loop through until NextDirectoryEntry(0) becomes zero -
       indicating that there are no more entries.
41      While NextDirectoryEntry(0)
42        ; If the directory entry is a file, not a folder.
43        If DirectoryEntryType(0) = #PB_DirectoryEntry_File
44
45          ; Add a new element to the list.
46          AddElement(Files())
47          ; And populate it with the properties of the file.
48          Files()\Name = DirectoryEntryName(0)
49          Files()\Size = DirectoryEntrySize(0)
50          Files()\Attributes = DirectoryEntryAttributes(0)
51          Files()\DateCreated = DirectoryEntryDate(0, #PB_Date_Created)
52          Files()\DateAccessed = DirectoryEntryDate(0, #PB_Date_Accessed)
53          Files()\DateModified = DirectoryEntryDate(0, #PB_Date_Modified)
54        EndIf
55      Wend
56      ; Close the directory.
57      FinishDirectory(0)
58    EndIf
59
60    ; Sort the list into ascending alphabetical order of file name.
61    SortStructuredList(Files(), #PB_Sort_Ascending,
       OffsetOf(FILEITEM\Name), #PB_String)
62
63    ; The interesting stuff starts to happen here...
64
65    ; This line defines a flag for the window attributes by OR-ing
       together the desired attribute constants.
66    Flags = #PB_Window_SystemMenu | #PB_Window_SizeGadget |
       #PB_Window_MinimizeGadget | #PB_Window_MaximizeGadget |
       #PB_Window_TitleBar
67
68    ; Open a GUI window.
69    OpenWindow(#WindowFiles, 50, 50, 450, 400, "File Properties", Flags)
70    ; A text gadget to show the name of the folder.
71    TextGadget(#Folder, 5, 40, 440, 25, Folder)
72    ; A list icon gadget to hold the file list and properties.
73    ListIconGadget(#Files, 5, 70, 440, 326, "#", 35)
74    ; Add columns to the ListIconGadget to hold each property.
75    AddGadgetColumn(#Files, 1, "Name", 200)
76    AddGadgetColumn(#Files, 2, "Created", 100)
77    AddGadgetColumn(#Files, 3, "Accessed", 100)
78    AddGadgetColumn(#Files, 4, "Modified", 100)
79    AddGadgetColumn(#Files, 5, "Attributes", 150)
80    AddGadgetColumn(#Files, 6, "Size", 100)
81
82    ; Load the files into the list view.
83    ForEach Files()
84      ; Display the item number and file name.
```

```
85      Num = StrU(ListIndex(Files()) + 1)
86
87      ; These lines convert the three date values to something more
        familiar.
88      Create = FormatDate("%dd/%mm/%yyyy", Files()\DateCreated)
89      Access = FormatDate("%dd/%mm/%yyyy", Files()\DateAccessed)
90      Modify = FormatDate("%dd/%mm/%yyyy", Files()\DateModified)
91
92      ; Convert the file size to a padded string the same as with the
        index value above,
93      ; but allow space for the maximum size of a quad.
94      Size = StrU(Files()\Size)
95
96      ; Convert the attributes to a string, for now.
97      Attrib = StrU(Files()\Attributes)
98
99      ; Build a row string.
100     ; The Line Feed character 'Chr(10)' tells the gadget to move to the
        next column.
101     Msg = Num + Chr(10) + Files()\Name + Chr(10) + Create + Chr(10) +
        Access + Chr(10) + Modify + Chr(10) + Attrib + Chr(10) + Size
102
103     ; Add the row to the list view gadget.
104     AddGadgetItem(#Files, -1, Msg)
105   Next Files()
106
107   ; This is the event loop for the window.
108   ; It will deal with all the user interaction events that we wish to
        use.
109
110   Repeat
111     ; Wait until a new window or gadget event occurs.
112     Event = WaitWindowEvent()
113     ; In programs with more than one form, which window did the event
        occur on.
114     EventWindow = EventWindow()
115     ; Which gadget did the event occur on.
116     EventGadget = EventGadget()
117     ; What sort of event occurred.
118     EventType = EventType()
119
120     ; Take some action.
121     Select Event
122
123       Case #PB_Event_Gadget
124         ; A gadget event occurred.
125         If EventGadget = #Folder
126         ElseIf EventGadget = #Files
127         EndIf
128
129       Case #PB_Event_CloseWindow
130         ; The window was closed.
131         If EventWindow = #WindowFiles
132           CloseWindow(#WindowFiles)
133           Break
134         EndIf
135
136     EndSelect
137
```

```
138     ; Go round and do it again.
139     ; In practice the loop isn't infinite because it can be stopped by
        clicking the window's Close button.
140   ForEver
```

At this point the application already has some useful features. However, it has some problems too:
1) You can't choose a folder to show.
2) You can't update the list contents without closing and restarting the program.
3) If you resize the window, the gadgets don't resize with it.
4) The attributes column is still not very useful.
We will revisit this program again later on to fix all these issues.

UserGuide Navigation

< Previous: Displaying text output (Console) || Overview || Next: Displaying graphics output & simple drawing >

Chapter 81

UserGuide - Input & Output

Every PureBasic application can communicate and interact with the user on different ways.
Thereby we distinguish between
a) the pure output of information
b) the interaction of the PureBasic application with the user, when user-input will be taken and the results will be outputted again.
It's not possible anymore, to use a simple "PRINT" command to output some things on the screen, like it was possible on DOS operating systems (OS) without a graphical user interface (GUI) years ago. Today such a GUI is always present, when you use an actual OS like Windows, Mac OSX or Linux.
For the output of information we have different possibilities:
- Debug window (only possible during programming with PureBasic)
- MessageRequester() (output of shorter text messages in a requester window)
- Files (for saving the results in a text-file, etc.)
- Console (for simple and almost non-graphic text output, most similar to earlier DOS times)
- Windows and gadgets (standard windows with GUI elements on the desktop of the OS, e.g. for applications)
- Screen (Output of text and graphics directly on the screen, e.g. for games)

To be able to **record and process input by the user** of the PureBasic application, the three last-mentioned output options have also the possibility to get user-input:
- in the console using Input()
- in the window using WaitWindowEvent() / WindowEvent() , which can get the events occurred in the window , like clicking on a button or the entered text in a StringGadget()
- in the graphics screen using the keyboard
- there is as well the possibility to get user-input using the InputRequester()

UserGuide Navigation

< Previous: Storing data in memory || Overview || Next: Displaying text output (Console) >

Chapter 82

UserGuide - Other Compiler keywords

~~<empty for now>~~

UserGuide Navigation

< Previous: Managing multiple windows || Overview || Next: Other Library functions >

Chapter 83

UserGuide - Other Library functions

~~<empty for now>~~

UserGuide Navigation

< Previous: Other Compiler keywords || Overview || Next: Advanced functions >

Chapter 84

UserGuide - Loops

Data, Events or many other things can also be processed using loops, which are always checked for a specific condition. Loops can be: Repeat : Until , Repeat : Forever , While : Wend , For : Next , ForEach : Next .
In this loop the counter A is increased by two each time, this loop will always perform the same number of iterations.

```
1   Define.i A
2   For A = 0 To 10 Step 2
3   ?? Debug A
4   Next A
```

This loop will increment the variable B by a random amount between 0 and 20 each time, until B exceeds 100. The number of iterations actually performed in the loop will vary depending on the random numbers. The check is performed at the start of the loop - so if the condition is already true, zero iterations may be performed. Take the ; away from the second line to see this happen.

```
1   Define.i B
2   ; B = 100
3   While B < 100
4 ??    B + Random(20)
5 ??    Debug B
6   Wend
```

This loop is very similar to the last except that the check is performed at the end of the loop. So one iteration, at least, will be performed. Again remove the ; from the second line to demonstrate.

```
1   Define.i C
2   ; C = 100
3   Repeat
4 ??    C + Random(20)
5 ??    Debug C
6   Until C > 99
```

This loop is infinite. It won't stop until you stop it (use the red X button on the IDE toolbar).

```
1   Define.i D
2   Repeat
3 ??    Debug D
4   ForEver
```

There is a special loop for working with lists and maps , it will iterate every member of the list (or map) in turn.

```
  NewList Fruit.s()

  AddElement(Fruit())
  Fruit() = "Banana"

  AddElement(Fruit())
  Fruit() = "Apple"

  AddElement(Fruit())
  Fruit() = "Pear"

  AddElement(Fruit())
  Fruit() = "Orange"

  ForEach Fruit()
??   Debug Fruit()
  Next Fruit()
```

UserGuide Navigation

< Previous: Decisions & Conditions || Overview || Next: String Manipulation >

Chapter 85

UserGuide - Memory access

Some PureBasic instructions, for example those from the Cipher library and many operating system API calls, require a pointer to a memory buffer as an argument rather than the data directly itself. PureBasic provides a number of instructions to manipulate memory buffers to facilitate this.
This example uses a buffer to read a file from disk into memory. It then converts the buffer content into a hexadecimal and text display in a ListIconGadget() as a simple hex viewer application.
An ExplorerListGadget() is used to display the contents of the user's home directory, initially, and to allow selection of a file. Two buttons are provided, one to display a file and another to clear the display.
The ListIcon Gadget is divided into nine columns , the first shows the base offset of each line in the list, the next show eight byte values offset from the base value and the ninth shows the string equivalent of these eight values.
Two pointers are used - the first (*Buffer) contains the memory address of the complete file. The second (*Byte), in the procedure "FileDisplay", demonstrates the use of pointer arithmetic and the Peek instruction to obtain individual values from within the buffer.
Finally, the "FileClose" procedure demonstrates the use of the FillMemory() instruction to overwrite the buffer's contents and FreeMemory() to de-allocate the memory buffer.

```
1   ;- Compiler Directives
2   EnableExplicit
3
4   ;- Constants
5   ; Window
6   Enumeration
7     #WindowHex
8   EndEnumeration
9
10  ; Gadgets
11  Enumeration
12    #GadgetFiles
13    #GadgetOpen
14    #GadgetClose
15    #GadgetHex
16  EndEnumeration
17
18  ;- Variables
19  Define.l Event, EventWindow, EventGadget, EventType, EventMenu
20  Define.l Length
21  Define.s File
22  Define *Buffer
23
24  ;- Declarations
25  Declare WindowCreate()
26  Declare WindowResize()
```

```
27  Declare FileClose()
28  Declare FileDisplay()
29  Declare FileRead()
30
31  ;- Implementation
32  Procedure WindowCreate()
33    ; Create the window.
34
35    Protected.l Col
36    Protected.s Label
37
38    If OpenWindow(#WindowHex, 50, 50, 500, 400, "Hex View",
      #PB_Window_SystemMenu | #PB_Window_SizeGadget |
      #PB_Window_MinimizeGadget | #PB_Window_TitleBar)
39
40      ; Set minimum window size.
41      WindowBounds(#WindowHex, 175, 175, #PB_Ignore, #PB_Ignore)
42
43      ; Create Explorer List and set to user's home directory.
44      ExplorerListGadget(#GadgetFiles, 5, 5, 490, 175,
      GetHomeDirectory())
45
46      ; Buttons.
47      ButtonGadget(#GadgetOpen, 5, 185, 80, 25, "Open")
48      ButtonGadget(#GadgetClose, 100, 185, 80, 25, "Close")
49
50      ; List Icon Gadget.
51      ListIconGadget(#GadgetHex, 5, 215, 490, 180, "Offset", 80,
      #PB_ListIcon_AlwaysShowSelection | #PB_ListIcon_GridLines |
      #PB_ListIcon_FullRowSelect)
52
53      ; Column Headings.
54      For Col = 0 To 7
55        Label = RSet(Hex(Col, #PB_Byte), 2, "0")
56        AddGadgetColumn(#GadgetHex, Col + 1, Label, 38)
57      Next Col
58      AddGadgetColumn(#GadgetHex, 9, "Text", 80)
59
60    EndIf
61
62  EndProcedure
63
64  Procedure WindowResize()
65    ; Resize gadgets to new window size.
66
67    Protected.l X, Y, W, H
68
69    ; Explorer List
70    W = WindowWidth(#WindowHex) - 10
71    H = (WindowHeight(#WindowHex) - 35) / 2
72    ResizeGadget(#GadgetFiles, #PB_Ignore, #PB_Ignore, W, H)
73
74    ; Buttons
75    Y = GadgetHeight(#GadgetFiles) + 10
76    ResizeGadget(#GadgetOpen, #PB_Ignore, Y, #PB_Ignore, #PB_Ignore)
77    ResizeGadget(#GadgetClose, #PB_Ignore, Y, #PB_Ignore, #PB_Ignore)
78
79    ; List Icon View
80    Y = (WindowHeight(#WindowHex) / 2) + 23
```

```
81      W = WindowWidth(#WindowHex) - 10
82      H = WindowHeight(#WindowHex) - (Y + 5)
83      ResizeGadget(#GadgetHex, #PB_Ignore, Y, W, H)
84
85    EndProcedure
86
87    Procedure FileClose()
88      ; Clear the list view and release the memory buffer.
89
90      Shared Length, *Buffer
91
92      ClearGadgetItems(#GadgetHex)
93      FillMemory(*Buffer, Length)
94      FreeMemory(*Buffer)
95
96    EndProcedure
97
98    Procedure FileDisplay()
99      ; Display the file buffer in the list view.
100
101     Shared Length, *Buffer
102
103     Protected *Byte
104     Protected Peek
105     Protected.l Rows, Cols, Offset
106     Protected.s OffsetString, Row, String
107
108     ; Clear current contents.
109     ClearGadgetItems(#GadgetHex)
110
111     ; Loop through rows.
112     For Rows = 0 To Length - 1 Step 8
113
114       ; Clear the text value for each row.
115       String = ""
116
117       ; Convert the offset value to a fixed length string.
118       Row = RSet(Hex(Rows, #PB_Long), 6, "0") + Chr(10)
119
120       ; Loop through columns.
121       For Cols = 0 To 7
122
123         ; Calculate the offset for the current column.
124         Offset = Rows + Cols
125
126         ; Compare the offset with the file length.
127         If Offset < Length
128           ; The offset is less than the length of the file.
129
130           ; Obtain the byte from the buffer.
131           *Byte = *Buffer + Offset
132           Peek = PeekB(*Byte)
133
134           ; Convert the byte to text.
135           Row + RSet(Hex(Peek, #PB_Byte), 2, "0") + Chr(10)
136
137           ; Add the character to the text version.
138           Select Peek
139
```

```
            Case 0 To 31, 127
              ; Unprintable characters.
              String + Chr(129)

            Default
              ; Printable characters.
              String + Chr(Peek)

          EndSelect

        Else
          ; The offset is greater than the length of the file.

          ; Add an empty column.
          Row + Chr(10)

        EndIf

      Next Cols

      ; Add the text version at the end of the hex columns.
      Row + String

      ; Add the completed row to the list view.
      AddGadgetItem(#GadgetHex, -1, Row)

    Next Rows

  EndProcedure

  Procedure FileRead()
    ; Read the file into the memory buffer.

    Shared Length, File, *Buffer

    Protected.b ReadByte
    Protected.l FileNumber, ReadLong, Size

    ; Stop if file is empty.
    If File = ""
      ProcedureReturn
    EndIf

    ; Stop if file size is invalid.
    Size = FileSize(File)
    If Size < 1
      ProcedureReturn
    EndIf

    ; Open the file.
    FileNumber = OpenFile(#PB_Any, File)
    Length = Lof(FileNumber)

    If File And Length

      ; Allocate a memory buffer to hold the file.
      *Buffer = AllocateMemory(Length)

      ; Read the file into the buffer.
```

```
199         Length = ReadData(FileNumber, *Buffer, Length)
200
201       EndIf
202
203       ; Close the file.
204       CloseFile(FileNumber)
205
206   EndProcedure
207
208   ;- Main
209   WindowCreate()
210
211   ;- Event Loop
212   Repeat
213
214       ; Obtain event parameters.
215       Event = WaitWindowEvent()
216       EventGadget = EventGadget()
217       EventType = EventType()
218       EventWindow = EventWindow()
219
220       ; Handle events.
221       Select Event
222
223         Case #PB_Event_Gadget
224           If EventGadget = #GadgetFiles
225             ; Do nothing.
226
227           ElseIf EventGadget = #GadgetOpen
228             File = GetGadgetText(#GadgetFiles) +
     GetGadgetItemText(#GadgetFiles, GetGadgetState(#GadgetFiles))
229             If FileSize(File) > 0
230               FileRead()
231               FileDisplay()
232             EndIf
233
234           ElseIf EventGadget = #GadgetClose
235             FileClose()
236
237           ElseIf EventGadget = #GadgetHex
238             ; Do nothing.
239
240           EndIf
241
242         Case #PB_Event_CloseWindow
243           If EventWindow = #WindowHex
244             CloseWindow(#WindowHex)
245             Break
246           EndIf
247
248         Case #PB_Event_SizeWindow
249           WindowResize()
250
251       EndSelect
252
253   ForEver
```

UserGuide Navigation

< Previous: Reading and writing files || Overview || Next: Dynamic numbering using #PB_Any >

Chapter 86

UserGuide - Overview

We start programming... this part of the PureBasic manual should guide you through some basic stuff, which you should learn while starting to program with PureBasic.
The following topics in this chapter should give you some ideas, where to start with PureBasic. It shouldn't replace any larger tutorial or the massive information, which can be found on the popular PureBasic forums . So the following information texts are short, but they include some "keywords" and links to further information, which are included in this reference manual. This chapter covers only a small part of the 1500+ commands available in PureBasic!

Topics in this chapter:

- First steps
- Variables and Processing of variables
- Constants
- Decisions & Conditions
- Loops
- String Manipulation
- Storing data in memory
- Input & Output
- Displaying text output (Console)
- Building a graphical user interface (GUI)
- Displaying graphics output & simple drawing
- Structuring code in Procedures
- Compiler directives (for different behavior on different OS)
- Reading and writing files
- Memory access
- Dynamic numbering using #PB_Any
- Managing multiple windows with different content
- Other Compiler keywords
- Other Library functions
- Advanced functions
- Some Tips & Tricks

UserGuide Navigation

Reference manual || Next: First steps >

Chapter 87

UserGuide - Dynamic numbering of windows and gadgets using #PB_Any

If youve looked at the help articles for the OpenWindow command or for any of the gadget creation commands (for example ButtonGadget()) or if you have experimented with the Form Designer tool, you may have noticed references to a special constant called #PB_Any . In this article were going to look into this a little further to find out why its so important.
So far all of our examples have used a group of constants, an enumeration , to identify a single window and each gadget on that window. This is fine in the simple programs weve demonstrated so far but presents a problem in more complex programs only one of each of these windows can exist at the same time.
So what happens if a program needs to provide more than one copy of a window? Maybe to have several files open at once or possibly to provide several different views of the same file.
This is where the special #PB_Any constant comes in. When this constant is used as the argument to the functions that support it, a unique reference number is automatically generated and returned as a result of the function.
Providing we keep track of all these references we can use this to our advantage. Organising this is a little more complex than the examples weve seen so far but the increased flexibility that it can provide makes this well worth the effort.
This example program provides a window upon which a regular polygon is drawn in blue on a circumscribing grey circle. A combo box is provided to allow a selection of polygons to be drawn. A menu is provided to allow the creation of new windows or the closing of the current window.
There are several things to notice about this program:

In the Enumerations section, note that there are only enumerations for the menu items . These will be shared on all the windows although each window will have its own menu bar.
In the Structures section, note that the POLYGONWINDOW structure contains four integer values and so provides a place to store references for a menu bar, a label, a combo box and an image plot.
In the Variables section note that in addition to the usual variables to receive event details, a map called ActiveWindows is created using the POLYGONWINDOW structure previously defined.
Look at the CreatePolygonWindow procedure .
When we create the window we capture the result of the OpenWindow() function in a variable called ThisWindow. We then convert this to a string value and use this as the key for a new map entry.
When we then create the menu , label , combo and image gadgets on the new window the references returned by these functions are stored in the map too.
Look at the ResizePolygonWindow procedure.
Notice how the value of EventWindow is passed in from the event loop into this procedure. This value is then used to retrieve child control references from the map and these references are then used to resize the gadgets.

Look at the DestroyPolygonWindow procedure.
Here the child control references are removed from the map when a window is closed. If the map size reaches zero there are no more open windows and DestroyPolygonWindow sets an event flag to tell the program to end.
Start the program up.
- Use the New Window menu item to open up two or three new polygon windows.
- Use the combo box in each one to select a different shape notice that each window works independently of all the others.
- Resize some of the windows notice that they can all be resized independently of each other and that the polygon resizes with the window too.

Finally, note that a map isnt the only way to achieve this effect, a List or an Array could be used to do the job too, if you prefer, although the code to implement these alternatives would need to be slightly different to that presented here because of the differences in the way those collections work.

```
1   ; Compiler Directives
2   EnableExplicit
3
4   ; Constants
5   CompilerIf Defined(Blue, #PB_Constant) = #False
6     #Blue = 16711680
7     #Gray = 8421504
8     #White = 16777215
9   CompilerEndIf
10
11  ;- Enumerations
12  ; The menu commands will be the same on all the windows.
13  Enumeration
14    #MenuNew
15    #MenuClose
16  EndEnumeration
17
18  ;- Structures
19  ; This structure will hold references to the unique elements of a
     window.
20  Structure POLYGONWINDOW
21    Menu.i
22    LabelSides.i
23    ComboSides.i
24    ImagePlot.i
25  EndStructure
26
27  ;- Variables
28
29  ; This map uses the previously defined structure to hold references
     for all the open windows.
30  NewMap ActiveWindows.POLYGONWINDOW()
31
32  ; Event variables.
33  Define.i Event, EventWindow, EventGadget, EventType, EventMenu,
     EventQuit
34  Define.s EventWindowKey
35
36  ; Implementation.
37  Procedure.i CreatePolygonWindow()
38    ; Creates a new window and gadgets, adding it and its child gadgets
     to the tracking map.
39    Shared ActiveWindows()
40    Protected.i ThisWindow
41    Protected.s ThisKey
```

```
42
43      ThisWindow = OpenWindow(#PB_Any, 50, 50, 300, 300, "Polygon",
        #PB_Window_SystemMenu | #PB_Window_SizeGadget |
        #PB_Window_MinimizeGadget | #PB_Window_TitleBar)
44
45      WindowBounds(ThisWindow, 250, 250, #PB_Ignore, #PB_Ignore)
46
47      If ThisWindow
48        ; Maps take a string value as key so convert the integer
        ThisWindow to a string.
49        ThisKey = StrU(ThisWindow)
50
51        ; Add a map element to hold the new gadget references.
52        AddMapElement(ActiveWindows(), ThisKey)
53
54        ; Create the menu bar.
55        ActiveWindows(ThisKey)\Menu = CreateMenu(#PB_Any,
        WindowID(ThisWindow))
56        MenuTitle("Window")
57        MenuItem(#MenuNew, "New Window")
58        MenuItem(#MenuClose, "Close Window")
59
60        ; Create the child gadgets and store their references in the map.
61        With ActiveWindows()
62          ; A label for the combo.
63          \LabelSides = TextGadget(#PB_Any, 5, 5, 150, 20, "Number of
        Sides:")
64
65          ; The Sides combo.
66          \ComboSides = ComboBoxGadget(#PB_Any, 160, 5, 100, 25)
67          AddGadgetItem(\ComboSides, 0, "Triangle")
68          AddGadgetItem(\ComboSides, 1, "Diamond")
69          AddGadgetItem(\ComboSides, 2, "Pentagon")
70          AddGadgetItem(\ComboSides, 3, "Hexagon")
71          AddGadgetItem(\ComboSides, 4, "Heptagon")
72          AddGadgetItem(\ComboSides, 5, "Octagon")
73          AddGadgetItem(\ComboSides, 6, "Nonagon")
74          AddGadgetItem(\ComboSides, 7, "Decagon")
75
76          ; Select Triangle.
77          SetGadgetState(\ComboSides, 0)
78
79          ; The visual image gadget on the window.
80          \ImagePlot = ImageGadget(#PB_Any, 5, 35, 290, 240, 0,
        #PB_Image_Border)
81        EndWith
82      EndIf
83
84      ; Return the reference to the new window.
85      ProcedureReturn ThisWindow
86    EndProcedure
87
88    Procedure DestroyPolygonWindow(Window.i)
89      ; Remove Window from the ActiveWindows map, close the window and
        set the quit flag, if appropriate.
90      Shared EventQuit, ActiveWindows()
91      Protected.s ThisKey
92
93      ; Convert the integer Window to a string.
```

```
94      ThisKey = StrU(Window)
95
96      ; Delete the map entry.
97      DeleteMapElement(ActiveWindows(), ThisKey)
98
99      ; Close the window.
100     CloseWindow(Window)
101
102     ; Check if there are still open windows.
103     If MapSize(ActiveWindows()) = 0
104       EventQuit = #True
105     EndIf
106   EndProcedure
107
108   Procedure.i ResizePolygonWindow(Window.i)
109     ; Resize the child gadgets on Window.
110     ; In practice only the ImageGadget needs to be resized in this
        example.
111     Shared ActiveWindows()
112     Protected.i ThisImage
113     Protected.i X, Y, W, H
114     Protected.s ThisKey
115
116     ; Obtain references to the affected gadgets from the map.
117     ThisKey = StrU(Window)
118     ThisImage = ActiveWindows(ThisKey)\ImagePlot
119
120     ; Resize gadgets.
121     W = WindowWidth(Window) - 15
122     H = WindowHeight(Window) - 70
123     ResizeGadget(ThisImage, #PB_Ignore, #PB_Ignore, W, H)
124   EndProcedure
125
126   Procedure PlotPolygon(Window.i)
127     ; Draw the polygon image and transfer it to the image gadget.
128     Shared ActiveWindows()
129     Protected.f Radius, OriginX, OriginY, StartX, StartY, EndX, EndY
130     Protected.i Sides, Vertex, Width, Height, ThisCombo, ThisImage,
        ThisPlot
131     Protected.s ThisKey
132
133     ; Check if the event is for the right window.
134     If Not IsWindow(Window) : ProcedureReturn : EndIf
135
136     ; Obtain references to the affected gadgets from the map.
137     ThisKey = StrU(Window)
138     ThisCombo = ActiveWindows(ThisKey)\ComboSides
139     ThisImage = ActiveWindows(ThisKey)\ImagePlot
140
141     ; Calculate dimensions and origin.
142     Sides = GetGadgetState(ThisCombo) + 3
143     Width = GadgetWidth(ThisImage) - 4
144     Height = GadgetHeight(ThisImage) - 4
145     OriginX = Width/2
146     OriginY = Height/2
147     If Width < Height
148       Radius = OriginX - 50
149     Else
150       Radius = OriginY - 50
```

```
    EndIf

    ; Create a new image.
    ThisPlot = CreateImage(#PB_Any, Width, Height)
    StartDrawing(ImageOutput(ThisPlot))

    ; Draw a white background.
    Box(0, 0, Width, Height, #White)

    ; Draw a gray circumscribing circle.
    Circle(OriginX, OriginY, Radius, #Gray)

    ; Draw the polygon.
    For Vertex = 0 To Sides

      ; Calculate side start point.
      StartX =  OriginX + (Radius * Cos(2 * #PI * Vertex/Sides))
      StartY = OriginY + (Radius * Sin(2 * #PI * Vertex/Sides))

      ; and end point.
      EndX = OriginX + (Radius * Cos(2 * #PI * (Vertex + 1)/Sides))
      EndY = OriginY + (Radius * Sin(2 * #PI * (Vertex + 1)/Sides))

      ; Draw the side in blue.
      LineXY(StartX, StartY, EndX, EndY, #Blue)

    Next Vertex

    ; Fill the polygon in blue
    FillArea(OriginX, OriginY, #Blue, #Blue)

    StopDrawing()

    ; Transfer the image contents to the visible gadget.
    SetGadgetState(ThisImage, ImageID(ThisPlot))

    ; Destroy the temporary image.
    FreeImage(ThisPlot)
  EndProcedure

  ;- Main

  ; Create the first window.
  EventWindow = CreatePolygonWindow()
  ResizePolygonWindow(EventWindow)
  PlotPolygon(EventWindow)

  ;- Event loop
  Repeat
    Event = WaitWindowEvent()
    EventWindow = EventWindow()
    EventWindowKey = StrU(EventWindow)
    EventGadget = EventGadget()
    EventType = EventType()
    EventMenu = EventMenu()

    Select Event
      Case #PB_Event_Gadget
        ; A gadget event has occurred.
```

```
210         If EventGadget = ActiveWindows(EventWindowKey)\LabelSides
211           ; Do nothing.
212 
213         ElseIf EventGadget = ActiveWindows(EventWindowKey)\ComboSides
214           ; If the combo box changes, redraw the image.
215           PlotPolygon(EventWindow)
216 
217           ; Update the windows title to reflect the new shape.
218           SetWindowTitle(EventWindow,
    GetGadgetText(ActiveWindows(EventWindowKey)\ComboSides))
219 
220         ElseIf EventGadget = ActiveWindows(EventWindowKey)\ImagePlot
221           ; Do nothing.
222 
223         EndIf
224 
225       Case #PB_Event_Menu
226         ; A menu event has occurred.
227         If EventMenu = #MenuNew
228           EventWindow = CreatePolygonWindow()
229           ResizePolygonWindow(EventWindow)
230           PlotPolygon(EventWindow)
231 
232         ElseIf EventMenu = #MenuClose
233           DestroyPolygonWindow(EventWindow)
234 
235         EndIf
236 
237       Case #PB_Event_Repaint
238         ; A window's content has been invalidated.
239         PlotPolygon(EventWindow)
240 
241       Case #PB_Event_SizeWindow
242         ; A window has been resized.
243         ResizePolygonWindow(EventWindow)
244         PlotPolygon(EventWindow)
245 
246       Case #PB_Event_CloseWindow
247         ; A window has been closed.
248         DestroyPolygonWindow(EventWindow)
249 
250     EndSelect
251 
252   Until EventQuit = #True
```

UserGuide Navigation

< Previous: Memory access || Overview || Next: Managing multiple windows >

Chapter 88

UserGuide - Managing multiple windows with different content

In the previous article we examined one way in which a program can support multiple instances of a single type of window . In this one we are going to extend this concept further developing a program that can support multiple instances of several different types of window, in this case three:

- The Button window contains a list view and two buttons labelled 'Add' and 'Remove'. When the 'Add' button is clicked a random integer is added to the list view, when the 'Remove' button is clicked the currently highlighted entry in the list view is removed.
- The Date window contains a list view and two buttons in the same way as the Button window but also contains a calendar gadget too, the window layout is altered to accommodate this additional control. When the 'Add' button is clicked, it is the currently selected date that is added to the list view. - The Track window contains two track bars , with a value between 0 and 100, and a string gadget . When the track bars are moved the string gadget is updated with the value of the second track bar subtracted from the first.

Each window contains a menu bar with items to create a new instance of any of the three supported window types or to close the current window.
Things to notice about this program are:
In the Structures section 4 structures are defined. The first, BASEWINDOW, defines a WindowClass value and a Menu value these values are common to each window type.
The remaining structures extend the BASEWINDOW structure with values for each of the unique controls that they require and which are not provided for by the BASEWINDOW structure.
In the Variables section note that again a map called ActiveWindows is created, however this time it is of integer type, it doesnt use any of the defined structures. There is a good reason for this, we need to store three different structure types to make this program work and we cant do this in a single map.
Also notice that *EventGadgets is defined using the BASEWINDOW structure.
Now look at the CreateButtonWindow procedure . As before we use #PB_Any to create the window and all the gadgets .
However this time the results are not stored directly in the ActiveWindows map. Instead we use the AllocateMemory function to allocate memory for a BUTTONWINDOW structure, we then store a pointer to this memory allocation in the ActiveWindows map. This is how we get around the problem of not being able to store all three of the different structures in the same map.
We also set the WindowClass value in the structure to #WindowClassButton to indicate which type of window, and therefore which type of structure, has been created we will need to know this later on.
There are two more CreateWindow procedures this time one for each class of the other window types. They work in a similar way to that described, differing only where the windows gadgets are different and setting a different value in WindowClass.
Similarly we provide DestroyWindow and ResizeWindow procedures to take care of these functions.
We also provide a new procedure EventsButtonWindow. This procedure knows what to do when any of the gadgets on the window are activated by the user. Similar procedures are provided for the other

window types too.
In all these procedures we use the ActiveWindows map to retrieve the pointer to the memory allocation. We can then use this pointer to retrieve the references to the actual controls that we need to work with in each of these procedures:

```
1  *ThisData = ActiveWindows(ThisKey)
2  *ThisData\ListView ...
```

Each procedure only knows how to handle one type of window so before we start work we check the WindowClass value to make sure that a window of the correct type has been supplied as the argument something like this:

```
1  If *ThisData\WindowClass <> #WindowClassButton
```

The event loop is a bit different too. For each event type there is a determination like this:

```
1  ; Use *EventGadgets\WindowClass to call the correct resize window
     procedure.
2  Select *EventGadgets\WindowClass ...
```

Although the memory allocations actually made by the CreateWindow procedures will be of the BUTTONWINDOW, DATEWINDOW or TRACKWINDOW type we can use *EventGadgets this way because it is defined as the BASEWINDOW type and BASEWINDOW is the ancestor structure to the other structures.
Providing we dont attempt to change any of the stored values using *EventGadgets and weve no reason to do so all should be well.
Finally, we dispatch the recorded event values in EventWindow, EventGadget, EventType straight through to the event procedures and let them worry about getting the jobs done.

```
1  ;- Constants
2  #DateFormat = "%dd/%mm/%yyyy"
3
4  ;- Enumerations
5  Enumeration
6    #WindowClassButton = 1
7    #WindowClassDate
8    #WindowClassTrack
9  EndEnumeration
10
11 ; The menu commands will be the same on all the windows.
12 Enumeration
13   #MenuNewButton
14   #MenuNewDate
15   #MenuNewTrack
16   #MenuClose
17 EndEnumeration
18
19 ;- Structures
20 Structure BASEWINDOW
21   WindowClass.i
22   Menu.i
23 EndStructure
24
25 Structure BUTTONWINDOW Extends BASEWINDOW
26   ListView.i
27   AddButton.i
28   RemoveButton.i
29 EndStructure
30
```

```
31  Structure DATEWINDOW Extends BASEWINDOW
32    Calendar.i
33    AddButton.i
34    RemoveButton.i
35    ListView.i
36  EndStructure
37
38  Structure TRACKWINDOW Extends BASEWINDOW
39    TrackBar1.i
40    TrackBar2.i
41    Label.i
42    Difference.i
43  EndStructure
44
45  ;- Variables
46  ; This map will track all the active windows as before, however as
     the structure for each window class
47  ; differs we will be storing pointers to structures in the map not
     the gadget references directly.
48  NewMap ActiveWindows.i()
49
50  ; These values will be used to give new windows a unique label.
51  Define.i Buttons, Dates, Tracks
52
53  ; Event variables.
54  ; Notice the type of *EventGadgets.
55  Define.i Event, EventWindow, EventGadget, EventType, EventMenu,
     EventQuit
56  Define.s EventWindowKey
57  Define *EventGadgets.BASEWINDOW
58
59  ;- Button Window
60  Procedure.i CreateButtonWindow()
61    ; Creates a new Button window and adds it to the tracking map,
62    ; allocates memory for gadget references, creates the gadgets
63    ; and stores these references in the memory structure.
64    Shared Buttons, ActiveWindows()
65    Protected *ThisData.BUTTONWINDOW
66    Protected.i ThisWindow
67    Protected.s ThisKey, ThisTitle
68
69    ; Set the window caption.
70    Buttons + 1
71    ThisTitle = "Button Window " + StrU(Buttons)
72
73    ; Open the window.
74    ThisWindow = OpenWindow(#PB_Any, 30, 30, 225, 300, ThisTitle,
     #PB_Window_SystemMenu | #PB_Window_SizeGadget |
     #PB_Window_MinimizeGadget | #PB_Window_TitleBar)
75
76    ; Check that the OpenWindow command worked.
77    If ThisWindow
78      ; Set minimum window dimensions.
79      WindowBounds(ThisWindow, 220, 100, #PB_Ignore, #PB_Ignore)
80
81      ; Convert the window reference to a string to use as the map key
     value.
82      ThisKey = StrU(ThisWindow)
83
```

```
84        ; Allocate memory to store the gadget references in.
85        *ThisData = AllocateMemory(SizeOf(BUTTONWINDOW))
86      EndIf
87
88      ; Check that the memory allocation worked.
89      If *ThisData
90        ; Store the window reference and memory pointer values in the map.
91        ActiveWindows(ThisKey) = *ThisData
92
93        ; Set the window class.
94        *ThisData\WindowClass = #WindowClassButton
95
96        ; Create the menu bar.
97        *ThisData\Menu = CreateMenu(#PB_Any, WindowID(ThisWindow))
98
99        ; If the menu creation worked, create the menu items.
100       If *ThisData\Menu
101         MenuTitle("Window")
102         MenuItem(#MenuNewButton, "New Button Window")
103         MenuItem(#MenuNewDate, "New Date Window")
104         MenuItem(#MenuNewTrack, "New Track Window")
105         MenuItem(#MenuClose, "Close Window")
106       EndIf
107
108       ; Create the window gadgets.
109       *ThisData\ListView = ListViewGadget(#PB_Any, 10, 10, 200, 225)
110       *ThisData\AddButton = ButtonGadget(#PB_Any, 15, 245, 90, 30,
      "Add")
111       *ThisData\RemoveButton = ButtonGadget(#PB_Any, 115, 245, 90, 30,
      "Remove")
112     Else
113       ; Memory allocation failed.
114       CloseWindow(ThisWindow)
115     EndIf
116
117     ; Set the return value.
118     If ThisWindow > 0 And *ThisData > 0
119       ; Return the reference to the new window.
120       ProcedureReturn ThisWindow
121     Else
122       ; Return 0
123       ProcedureReturn 0
124     EndIf
125   EndProcedure
126
127   Procedure.i DestroyButtonWindow(Window.i)
128     ; Remove Window from the ActiveWindows map, release the allocated
      memory,
129     ; close the window and set the quit flag, if appropriate.
130     Shared EventQuit, ActiveWindows()
131     Protected *ThisData.BUTTONWINDOW
132     Protected.s ThisKey
133
134     ; Convert the integer Window to a string.
135     ThisKey = StrU(Window)
136
137     ; Obtain the reference structure pointer.
138     *ThisData = ActiveWindows(ThisKey)
139
```

```
140     ; Check that a valid pointer was obtained, if not stop.
141     If *ThisData = 0
142       ProcedureReturn #False
143     EndIf
144 
145     ; Check that it is the correct window type, if not stop.
146     If *ThisData\WindowClass <> #WindowClassButton
147       ProcedureReturn #False
148     EndIf
149 
150     ; Release the memory allocation.
151     FreeMemory(*ThisData)
152 
153     ; Delete the map entry.
154     DeleteMapElement(ActiveWindows(), ThisKey)
155 
156     ; Close the window.
157     CloseWindow(Window)
158 
159     ; Check if there are still open windows.
160     If MapSize(ActiveWindows()) = 0
161       EventQuit = #True
162     EndIf
163 
164     ; Set the successful return value.
165     ProcedureReturn #True
166   EndProcedure
167 
168   Procedure.i ResizeButtonWindow(Window.i)
169     ; Resize the child gadgets on Window.
170     Shared ActiveWindows()
171     Protected *ThisData.BUTTONWINDOW
172     Protected.i X, Y, W, H
173     Protected.s ThisKey
174 
175     ; Obtain the reference structure pointer.
176     ThisKey = StrU(Window)
177     *ThisData = ActiveWindows(ThisKey)
178 
179     ; Check that a valid pointer was obtained, if not stop.
180     If *ThisData = 0
181       ProcedureReturn #False
182     EndIf
183 
184     ; Check that it is the correct window type, if not stop.
185     If *ThisData\WindowClass <> #WindowClassButton
186       ProcedureReturn #False
187     EndIf
188 
189     ; Resize list view.
190     W = WindowWidth(Window) - 25
191     H = WindowHeight(Window) - 85
192     ResizeGadget(*ThisData\ListView, #PB_Ignore, #PB_Ignore, W, H)
193 
194     ; Recenter buttons.
195     X = WindowWidth(Window)/2 - 95
196     Y = WindowHeight(Window) - 65
197     ResizeGadget(*ThisData\AddButton, X, Y, #PB_Ignore, #PB_Ignore)
198 
```

```
    X = WindowWidth(Window)/2 + 5
    ResizeGadget(*ThisData\RemoveButton, X, Y, #PB_Ignore, #PB_Ignore)

    ProcedureReturn #True
  EndProcedure

  Procedure.i EventsButtonWindow(Window, Gadget, Type)
    ; Handle events for a button window.
    Shared Buttons, ActiveWindows()
    Protected *ThisData.BUTTONWINDOW
    Protected.i NewValue, Index
    Protected.s ThisKey

    ; Convert the integer Window to a string.
    ThisKey = StrU(Window)

    ; Obtain the reference structure pointer.
    *ThisData = ActiveWindows(ThisKey)

    ; Check that a valid pointer was obtained, if not stop.
    If *ThisData = 0
      ProcedureReturn #False
    EndIf

    ; Check that it is the correct window type, if not stop.
    If *ThisData\WindowClass <> #WindowClassButton
      ProcedureReturn #False
    EndIf

    Select Gadget
      Case *ThisData\AddButton
        NewValue = Random(2147483647)
        AddGadgetItem(*ThisData\ListView, -1, StrU(NewValue))

      Case *ThisData\RemoveButton
        Index = GetGadgetState(*ThisData\ListView)
        If Index >= 0 And Index <= CountGadgetItems(*ThisData\ListView)
          RemoveGadgetItem(*ThisData\ListView, Index)
        EndIf

      Case *ThisData\ListView
        ; Do nothing.
    EndSelect
  EndProcedure

  ;- Date Window
  Procedure.i CreateDateWindow()
    ; Creates a new Date window and adds it to the tracking map,
    ; allocates memory for gadget references, creates the gadgets
    ; and stores these references in the memory Structure.
    Shared Dates, ActiveWindows()
    Protected *ThisData.DATEWINDOW
    Protected.i ThisWindow
    Protected.s ThisKey, ThisTitle

    Dates + 1
    ThisTitle = "Date Window " + StrU(Dates)
    ThisWindow = OpenWindow(#PB_Any, 30, 30, 310, 420, ThisTitle ,
   #PB_Window_SystemMenu | #PB_Window_SizeGadget |
```

```
    #PB_Window_MinimizeGadget | #PB_Window_TitleBar)

    ; Check that the OpenWindow command worked.
    If ThisWindow
      ; Set minimum window dimensions.
      WindowBounds(ThisWindow, 310, 245, #PB_Ignore, #PB_Ignore)

      ; Convert the window reference to a string to use as the map key value.
      ThisKey = StrU(ThisWindow)

      ; Allocate memory to store the gadget references in.
      *ThisData = AllocateMemory(SizeOf(DATEWINDOW))
    EndIf

    ; Check that the memory allocation worked.
    If *ThisData
      ; Store the window reference and memory pointer values in the map.
      ActiveWindows(ThisKey) = *ThisData

      ; Set the window class.
      *ThisData\WindowClass = #WindowClassDate

      ; Create the menu bar.
      *ThisData\Menu = CreateMenu(#PB_Any, WindowID(ThisWindow))

      ; If the menu creation worked, create the menu items.
      If *ThisData\Menu
        MenuTitle("Window")
        MenuItem(#MenuNewButton, "New Button Window")
        MenuItem(#MenuNewDate, "New Date Window")
        MenuItem(#MenuNewTrack, "New Track Window")
        MenuItem(#MenuClose, "Close Window")
      EndIf

      ; Create the window gadgets.
      *ThisData\Calendar = CalendarGadget(#PB_Any, 10, 10, 182, 162)
      *ThisData\AddButton = ButtonGadget(#PB_Any, 210, 10, 90, 30, "Add")
      *ThisData\RemoveButton = ButtonGadget(#PB_Any, 210, 45, 90, 30, "Remove")
      *ThisData\ListView = ListViewGadget(#PB_Any, 10, 187, 290, 200)
    Else
      ; Memory allocation failed.
      CloseWindow(ThisWindow)
    EndIf

    ; Set the return value.
    If ThisWindow > 0 And *ThisData > 0
      ; Return the reference to the new window.
      ProcedureReturn ThisWindow
    Else
      ; Return 0
      ProcedureReturn 0
    EndIf

  EndProcedure

  Procedure.i DestroyDateWindow(Window.i)
```

```
312     ; Remove Window from the ActiveWindows map, release the allocated
        memory,
313     ; close the window and set the quit flag, if appropriate.
314     Shared EventQuit, ActiveWindows()
315     Protected *ThisData.DATEWINDOW
316     Protected.s ThisKey
317
318     ; Convert the integer Window to a string.
319     ThisKey = StrU(Window)
320
321     ; Obtain the reference structure pointer.
322     *ThisData = ActiveWindows(ThisKey)
323
324     ; Check that a valid pointer was obtained.
325     If *ThisData = 0
326       ProcedureReturn #False
327     EndIf
328
329     ; Check that it is the correct window type, if not stop as this
        procedure can't destroy this window.
330     If *ThisData\WindowClass <> #WindowClassDate
331       ProcedureReturn #False
332     EndIf
333
334     ; Release the memory allocation.
335     FreeMemory(*ThisData)
336
337     ; Delete the map entry.
338     DeleteMapElement(ActiveWindows(), ThisKey)
339
340     ; Close the window.
341     CloseWindow(Window)
342
343     ; Check if there are still open windows.
344     If MapSize(ActiveWindows()) = 0
345       EventQuit = #True
346     EndIf
347
348     ; Set the successful return value.
349     ProcedureReturn #True
350   EndProcedure
351
352   Procedure.i ResizeDateWindow(Window.i)
353     ; Resize the child gadgets on Window.
354     Shared ActiveWindows()
355     Protected *ThisData.DATEWINDOW
356     Protected.i X, Y, W, H
357     Protected.s ThisKey
358
359     ; Obtain the reference structure pointer.
360     ThisKey = StrU(Window)
361     *ThisData = ActiveWindows(ThisKey)
362
363     ; Check that a valid pointer was obtained, if not stop.
364     If *ThisData = 0
365       ProcedureReturn #False
366     EndIf
367
368     ; Check that it is the correct window type, if not stop.
```

```
    If *ThisData\WindowClass <> #WindowClassDate
      ProcedureReturn #False
    EndIf

    ; Resize list view.
    W = WindowWidth(Window) - 20
    H = WindowHeight(Window) - 220
    ResizeGadget(*ThisData\ListView, #PB_Ignore, #PB_Ignore, W, H)

    ProcedureReturn #True
  EndProcedure

  Procedure.i EventsDateWindow(Window, Gadget, Type)
    ; Handle events for a Date window.
    Shared Buttons, ActiveWindows()
    Protected *ThisData.DATEWINDOW
    Protected.i NewValue, Index
    Protected.s ThisKey

    ; Convert the integer Window to a string.
    ThisKey = StrU(Window)

    ; Obtain the reference structure pointer.
    *ThisData = ActiveWindows(ThisKey)

    ; Check that a valid pointer was obtained, if not stop.
    If *ThisData = 0
      ProcedureReturn #False
    EndIf

    ; Check that it is the correct window type, if not stop.
    If *ThisData\WindowClass <> #WindowClassDate
      ProcedureReturn #False
    EndIf

    Select Gadget
      Case *ThisData\AddButton
        NewValue = GetGadgetState(*ThisData\Calendar)
        AddGadgetItem(*ThisData\ListView, -1, FormatDate(#DateFormat,
 NewValue))

      Case *ThisData\RemoveButton
        Index = GetGadgetState(*ThisData\ListView)
        If Index >= 0 And Index <= CountGadgetItems(*ThisData\ListView)
          RemoveGadgetItem(*ThisData\ListView, Index)
        EndIf

      Case *ThisData\Calendar, *ThisData\ListView
        ; Do nothing.
    EndSelect
  EndProcedure

  ;- Track Window
  Procedure.i CreateTrackWindow()
    ; Creates a new Track window and adds it to the tracking map,
    ; allocates memory for gadget references, creates the gadgets
    ; and stores these references in the memory Structure.
    Shared Tracks, ActiveWindows()
    Protected *ThisData.TRACKWINDOW
```

```
427     Protected.i ThisWindow, ThisSum
428     Protected.s ThisKey, ThisTitle
429
430     Tracks + 1
431     ThisTitle = "Track Bar Window " + StrU(Tracks)
432     ThisWindow = OpenWindow(#PB_Any, 30, 30, 398, 130, ThisTitle,
        #PB_Window_SystemMenu | #PB_Window_SizeGadget |
        #PB_Window_MinimizeGadget | #PB_Window_TitleBar)
433
434     ; Check that the OpenWindow command worked.
435     If ThisWindow
436       ; Set minimum window dimensions.
437       WindowBounds(ThisWindow, 135, 130, #PB_Ignore, 130)
438
439       ; Convert the window reference to a string to use as the map key
        value.
440       ThisKey = StrU(ThisWindow)
441
442       ; Allocate memory to store the gadget references in.
443       *ThisData = AllocateMemory(SizeOf(TRACKWINDOW))
444     EndIf
445
446     ; Check that the memory allocation worked.
447     If *ThisData
448       ; Store the window reference and memory pointer values in the map.
449       ActiveWindows(ThisKey) = *ThisData
450
451       ; Set the window class.
452       *ThisData\WindowClass = #WindowClassTrack
453
454       ; Create the menu bar.
455       *ThisData\Menu = CreateMenu(#PB_Any, WindowID(ThisWindow))
456
457       ; If the menu creation worked, create the menu items.
458       If *ThisData\Menu
459         MenuTitle("Window")
460         MenuItem(#MenuNewButton, "New Button Window")
461         MenuItem(#MenuNewDate, "New Date Window")
462         MenuItem(#MenuNewTrack, "New Track Window")
463         MenuItem(#MenuClose, "Close Window")
464       EndIf
465
466       ; Create the window gadgets.
467       *ThisData\TrackBar1 = TrackBarGadget(#PB_Any, 10, 10, 375, 25, 0,
        100, #PB_TrackBar_Ticks)
468       *ThisData\TrackBar2 = TrackBarGadget(#PB_Any, 10, 40, 375, 25, 0,
        100, #PB_TrackBar_Ticks)
469       *ThisData\Label = TextGadget(#PB_Any, 10, 75, 80, 25,
        "Difference:")
470       *ThisData\Difference = StringGadget(#PB_Any, 90, 75, 290, 25,
        "0", #PB_String_ReadOnly)
471     Else
472       ; Memory allocation failed.
473       CloseWindow(ThisWindow)
474     EndIf
475
476     ; Set the return value.
477     If ThisWindow > 0 And *ThisData > 0
478       ; Return the reference to the new window.
```

```
479       ProcedureReturn ThisWindow
480     Else
481       ; Return 0
482       ProcedureReturn 0
483     EndIf
484   EndProcedure
485
486   Procedure.i DestroyTrackWindow(Window.i)
487     ; Remove Window from the ActiveWindows map, release the allocated
      memory,
488     ; close the window and set the quit flag, if appropriate.
489     Shared EventQuit, ActiveWindows()
490     Protected *ThisData.DATEWINDOW
491     Protected.s ThisKey
492
493     ; Convert the integer Window to a string.
494     ThisKey = StrU(Window)
495
496     ; Obtain the reference structure pointer.
497     *ThisData = ActiveWindows(ThisKey)
498
499     ; Check that a valid pointer was obtained.
500     If *ThisData = 0
501       ProcedureReturn #False
502     EndIf
503
504     ; Check that it is the correct window type, if not stop as this
      procedure can't destroy this window.
505     If *ThisData\WindowClass <> #WindowClassTrack
506       ProcedureReturn #False
507     EndIf
508
509     ; Release the memory allocation.
510     FreeMemory(*ThisData)
511
512     ; Delete the map entry.
513     DeleteMapElement(ActiveWindows(), ThisKey)
514
515     ; Close the window.
516     CloseWindow(Window)
517
518     ; Check if there are still open windows.
519     If MapSize(ActiveWindows()) = 0
520       EventQuit = #True
521     EndIf
522
523     ; Set the successful return value.
524     ProcedureReturn #True
525   EndProcedure
526
527   Procedure.i ResizeTrackWindow(Window.i)
528     ; Resize the child gadgets on Window.
529     Shared ActiveWindows()
530     Protected *ThisData.TRACKWINDOW
531     Protected.i X, Y, W, H
532     Protected.s ThisKey
533
534     ; Obtain the reference structure pointer.
535     ThisKey = StrU(Window)
```

```
536     *ThisData = ActiveWindows(ThisKey)
537
538     ; Check that a valid pointer was obtained, if not stop.
539     If *ThisData = 0
540       ProcedureReturn #False
541     EndIf
542
543     ; Check that it is the correct window type, if not stop.
544     If *ThisData\WindowClass <> #WindowClassTrack
545       ProcedureReturn #False
546     EndIf
547
548     ; Resize track bars.
549     W = WindowWidth(Window) - 20
550     ResizeGadget(*ThisData\TrackBar1, #PB_Ignore, #PB_Ignore, W,
      #PB_Ignore)
551     ResizeGadget(*ThisData\TrackBar2, #PB_Ignore, #PB_Ignore, W,
      #PB_Ignore)
552
553     ; Resize string.
554     W = WindowWidth(Window) - 110
555     ResizeGadget(*ThisData\Difference, #PB_Ignore, #PB_Ignore, W,
      #PB_Ignore)
556
557     ProcedureReturn #True
558   EndProcedure
559
560   Procedure.i EventsTrackWindow(Window, Gadget, Type)
561     ; Handle events for a Track window.
562     Shared Buttons, ActiveWindows()
563     Protected *ThisData.TRACKWINDOW
564     Protected.i NewValue
565     Protected.s ThisKey
566
567     ; Convert the integer Window to a string.
568     ThisKey = StrU(Window)
569
570     ; Obtain the reference structure pointer.
571     *ThisData = ActiveWindows(ThisKey)
572
573     ; Check that a valid pointer was obtained, if not stop.
574     If *ThisData = 0
575       ProcedureReturn #False
576     EndIf
577
578     ; Check that it is the correct window type, if not stop.
579     If *ThisData\WindowClass <> #WindowClassTrack
580       ProcedureReturn #False
581     EndIf
582
583     Select Gadget
584       Case *ThisData\TrackBar1, *ThisData\TrackBar2
585         NewValue = GetGadgetState(*ThisData\TrackBar1) -
      GetGadgetState(*ThisData\TrackBar2)
586         SetGadgetText(*ThisData\Difference, Str(NewValue))
587
588       Case *ThisData\Label, *ThisData\Difference
589         ; Do nothing.
590     EndSelect
```

```
591   EndProcedure
592
593   ;- Main
594
595   ; Create the first window.
596   EventWindow = CreateButtonWindow()
597   ResizeButtonWindow(EventWindow)
598
599   ;- Event loop
600   Repeat
601     Event = WaitWindowEvent()
602     EventWindow = EventWindow()
603     EventWindowKey = StrU(EventWindow)
604     EventGadget = EventGadget()
605     EventType = EventType()
606     EventMenu = EventMenu()
607     *EventGadgets = ActiveWindows(EventWindowKey)
608
609     Select Event
610       Case #PB_Event_Gadget
611         ; Check that a valid pointer was obtained.
612         If *EventGadgets > 0
613           ; Use *EventGadgets\WindowClass to dispatch events to the
      correct event procedure.
614           Select *EventGadgets\WindowClass
615             Case #WindowClassButton
616               EventsButtonWindow(EventWindow, EventGadget, EventType)
617
618             Case #WindowClassDate
619               EventsDateWindow(EventWindow, EventGadget, EventType)
620
621             Case #WindowClassTrack
622               EventsTrackWindow(EventWindow, EventGadget, EventType)
623
624             Default
625               ; Do nothing
626           EndSelect
627         EndIf
628
629       Case #PB_Event_Menu
630         Select EventMenu
631           Case #MenuNewButton
632             EventWindow = CreateButtonWindow()
633             If EventWindow > 0
634               ResizeButtonWindow(EventWindow)
635             EndIf
636
637           Case #MenuNewDate
638             EventWindow = CreateDateWindow()
639             If EventWindow > 0
640               ResizeDateWindow(EventWindow)
641             EndIf
642
643           Case #MenuNewTrack
644             EventWindow = CreateTrackWindow()
645             If EventWindow > 0
646               ResizeTrackWindow(EventWindow)
647             EndIf
648
```

```
649           Case #MenuClose
650             ; Check that a valid pointer was obtained.
651             If *EventGadgets > 0
652               ; Use *EventGadgets\WindowClass to call the correct
    destroy window procedure.
653               Select *EventGadgets\WindowClass
654                 Case #WindowClassButton
655                   DestroyButtonWindow(EventWindow)
656
657                 Case #WindowClassDate
658                   DestroyDateWindow(EventWindow)
659
660                 Case #WindowClassTrack
661                   DestroyTrackWindow(EventWindow)
662
663                 Default
664                   ; Do nothing
665               EndSelect
666             EndIf
667         EndSelect
668
669       Case #PB_Event_CloseWindow
670         ; Check that a valid pointer was obtained.
671         If *EventGadgets > 0
672           ; Use *EventGadgets\WindowClass to call the correct destroy
    window procedure.
673           Select *EventGadgets\WindowClass
674             Case #WindowClassButton
675               DestroyButtonWindow(EventWindow)
676
677             Case #WindowClassDate
678               DestroyDateWindow(EventWindow)
679
680             Case #WindowClassTrack
681               DestroyTrackWindow(EventWindow)
682
683             Default
684               ; Do nothing
685           EndSelect
686         EndIf
687
688       Case #PB_Event_SizeWindow
689         If *EventGadgets > 0
690           ; Use *EventGadgets\WindowClass to call the correct resize
    window procedure.
691           Select *EventGadgets\WindowClass
692             Case #WindowClassButton
693               ResizeButtonWindow(EventWindow)
694
695             Case #WindowClassDate
696               ResizeDateWindow(EventWindow)
697
698             Case #WindowClassTrack
699               ResizeTrackWindow(EventWindow)
700
701             Default
702               ; Do nothing
703           EndSelect
704         EndIf
```

```
705
706     EndSelect
707
708   Until EventQuit = #True
```

UserGuide Navigation

< Previous: Dynamic numbering of windows and gadgets || Overview || Next: Other Compiler keywords >

Chapter 89

UserGuide - Structuring code in Procedures

We're going to revisit the file properties example again. This time to introduce procedures and to address some of the limitations identified in the program in previous items.

```
1   ; The structure for file information as before.
2   Structure FILEITEM
3     Name.s
4     Attributes.i
5     Size.q
6     DateCreated.i
7     DateAccessed.i
8     DateModified.i
9   EndStructure
10
11  ; This is a constant to identify the window.
12  Enumeration
13    #WindowFiles
14  EndEnumeration
15
16  ; This is an enumeration to identify controls that will appear on the
     window.
17  Enumeration
18    #FolderButton
19    #UpdateButton
20    #FolderText
21    #FilesList
22  EndEnumeration
23
24  Procedure FilesExamine(Folder.s, List Files.FILEITEM())
25    ; Obtains file properties from Folder into Files.
26
27    Protected.l Result
28
29    ; Clear current contents.
30    ClearList(Files())
31
32    ; Open the directory to enumerate its contents.
33    Result = ExamineDirectory(0, Folder, "*.*")
34
35    ; If this is ok, begin enumeration of entries.
36    If Result
```

```
37        ; Loop through until NextDirectoryEntry(0) becomes zero -
      indicating that there are no more entries.
38        While NextDirectoryEntry(0)
39          ; If the directory entry is a file, not a folder.
40          If DirectoryEntryType(0) = #PB_DirectoryEntry_File
41            ; Add a new element to the list.
42            AddElement(Files())
43            ; And populate it with the properties of the file.
44            Files()\Name = DirectoryEntryName(0)
45            Files()\Size = DirectoryEntrySize(0)
46            Files()\Attributes = DirectoryEntryAttributes(0)
47            Files()\DateCreated = DirectoryEntryDate(0, #PB_Date_Created)
48            Files()\DateAccessed = DirectoryEntryDate(0,
      #PB_Date_Accessed)
49            Files()\DateModified = DirectoryEntryDate(0,
      #PB_Date_Modified)
50          EndIf
51        Wend
52
53        ; Close the directory.
54        FinishDirectory(0)
55      EndIf
56
57      ; Sort the list into ascending alphabetical order of file name.
58      SortStructuredList(Files(), #PB_Sort_Ascending,
      OffsetOf(FILEITEM\Name), #PB_String)
59    EndProcedure
60
61    Procedure.s FolderSelect(Folder.s)
62      ; Displays a path requester and returns the new path, or the old
      one if the requester is cancelled.
63      Protected.s SelectedPath
64
65      SelectedPath = PathRequester("Choose a folder.", Folder)
66
67      If SelectedPath = ""
68        SelectedPath = Folder
69      EndIf
70
71      ProcedureReturn SelectedPath
72    EndProcedure
73
74    Procedure LabelUpdate(Folder.s)
75      ; Updates the folder label.
76      SetGadgetText(#FolderText, Folder)
77    EndProcedure
78
79    Procedure ListLoad(ListView.l, List Files.FILEITEM())
80      ; Load the files properties from list Files() into the list view
      #FilesList.
81      Protected.s Access, Attrib, Create, Folder, Modify, Msg, Num, Size
82
83      ; Remove previous contents.
84      ClearGadgetItems(ListView)
85
86      ForEach Files()
87        ; Display the item number and file name.
88        Num = StrU(ListIndex(Files()) + 1)
89
```

```
90        ; These lines convert the three date values to something more
      familiar.
91        Create = FormatDate("%dd/%mm/%yyyy", Files()\DateCreated)
92        Access = FormatDate("%dd/%mm/%yyyy", Files()\DateAccessed)
93        Modify = FormatDate("%dd/%mm/%yyyy", Files()\DateModified)
94
95        ; Convert the file size to a padded string the same as with the
      index value above,
96        ; but allow space for the maximum size of a quad.
97        Size = StrU(Files()\Size)
98
99        ; Convert the attributes to a string, for now.
100       Attrib = StrU(Files()\Attributes)
101
102       ; Build a row string.
103       ; The Line Feed character 'Chr(10)' tells the gadget to move to
      the next column.
104       Msg = Num + Chr(10) + Files()\Name + Chr(10) + Create + Chr(10) +
      Access + Chr(10) + Modify + Chr(10) + Attrib + Chr(10) + Size
105
106       ; Add the row to the list view gadget.
107       AddGadgetItem(#FilesList, -1, Msg)
108     Next Files()
109   EndProcedure
110
111   Procedure WindowCreate()
112     ; Creates the wdwFiles window.
113     Protected Flags
114
115     ; This line defines a flag for the window attributes by OR-ing
      together the desired attribute constants.
116     Flags = #PB_Window_SystemMenu | #PB_Window_SizeGadget |
      #PB_Window_MinimizeGadget | #PB_Window_MaximizeGadget|
      #PB_Window_TitleBar
117
118     ; Open a window.
119     OpenWindow(#WindowFiles, 50, 50, 450, 400, "File Properties", Flags)
120     ; A button to choose a folder.
121     ButtonGadget(#FolderButton, 5, 5, 100, 30, "Select Folder")
122     ; A button to update the list.
123     ButtonGadget(#UpdateButton, 112, 5, 100, 30, "Update List")
124     ; A text gadget to show the name of the folder.
125     TextGadget(#FolderText, 5, 40, 400, 25, "")
126     ; A list icon gadget to hold the file list and properties.
127     ListIconGadget(#FilesList, 5, 70, 400, 326, "#", 35)
128     ; Add columns to the ListIconGadget to hold each property.
129     AddGadgetColumn(#FilesList, 1, "Name", 200)
130     AddGadgetColumn(#FilesList, 2, "Created", 100)
131     AddGadgetColumn(#FilesList, 3, "Accessed", 100)
132     AddGadgetColumn(#FilesList, 4, "Modified", 100)
133     AddGadgetColumn(#FilesList, 5, "Attributes", 150)
134     AddGadgetColumn(#FilesList, 6, "Size", 100)
135   EndProcedure
136
137   Procedure WindowDestroy()
138     ; Closes the window.
139     ; If necessary, you could do other tidying up jobs here too.
140     CloseWindow(#WindowFiles)
141   EndProcedure
```

```
142
143   Procedure WindowResize()
144     ; Resizes window gadgets to match the window size.
145     ResizeGadget(#FolderText, #PB_Ignore, #PB_Ignore,
      WindowWidth(#WindowFiles) - 10, #PB_Ignore)
146     ResizeGadget(#FilesList, #PB_Ignore, #PB_Ignore,
      WindowWidth(#WindowFiles) - 10, WindowHeight(#WindowFiles) - 74)
147   EndProcedure
148
149   ;- Main
150   ; Now we define a list of files using the structure previously
       specified.
151   NewList Files.FILEITEM()
152
153   ; And some working variables to make things happen.
154   Define.s Folder
155   Define.l Event, EventWindow, EventGadget, EventType, EventMenu
156
157   ; This function gets the home directory for the logged on user.
158   Folder = GetHomeDirectory()
159
160   ; Create the window and set the initial contents.
161   WindowCreate()
162   WindowResize()
163   LabelUpdate(Folder)
164   FilesExamine(Folder, Files())
165   ListLoad(#FilesList, Files())
166
167   ;- Event Loop
168   Repeat
169     ; Wait until a new window or gadget event occurs.
170     Event = WaitWindowEvent()
171     EventWindow = EventWindow()
172     EventGadget = EventGadget()
173     EventType = EventType()
174
175     ; Take some action.
176     Select Event
177       Case #PB_Event_Gadget
178         ; A gadget event occurred.
179         If EventGadget = #FolderButton
180           ; The folder button was clicked.
181           Folder = FolderSelect(Folder)
182           LabelUpdate(Folder)
183           FilesExamine(Folder, Files())
184           ListLoad(#FilesList, Files())
185
186         ElseIf EventGadget = #UpdateButton
187           ; The update button was clicked.
188           FilesExamine(Folder, Files())
189           ListLoad(#FilesList, Files())
190
191         ElseIf EventGadget = #FolderText
192           ; Do nothing here.
193
194         ElseIf EventGadget = #FilesList
195           ; Do nothing here.
196
197         EndIf
```

```
198
199         Case #PB_Event_SizeWindow
200           ; The window was moved or resized.
201           If EventWindow = #WindowFiles
202             WindowResize()
203           EndIf
204
205         Case #PB_Event_CloseWindow
206           ; The window was closed.
207           If EventWindow = #WindowFiles
208             WindowDestroy()
209             Break
210
211           EndIf
212       EndSelect
213     ForEver
```

Previously, we mentioned four limitations to this program. This new version uses procedures to address three of them.
1) You couldn't choose a folder to show.
The "FolderSelect" procedure shows a path requester to allow the user to select a folder. The variable "Folder" is updated with the result of this procedure. The button also calls "LabelUpdate", "FilesExamine" and "ListLoad" to display the contents of the new folder in the window .
2) You can't update the list contents without closing and restarting the program.
Now, when the "Update List" button is clicked, "FilesExamine" and "ListLoad" are called again to update the display.
3) If you resize the window, the gadgets don't resize with it.
The "WindowResize" procedure is called in the event loop to resize the gadgets when the form is resized. Also, although this program didn't really need to, this procedure is called after calling "WindowCreate" to make sure the gadgets are the right size initially.
Notice how several of the procedures are called more than once to perform similar but not identical functions. This improves the efficiency of the program.
We have one final limitation to overcome in a later item.

UserGuide Navigation

< Previous: Displaying graphics output & simple drawing || Overview || Next: Compiler directives >

Chapter 90

UserGuide - String Manipulation

The following example shows step by step the different commands of the string library - their purpose and their correct use.

```
1   Define.s String1, String2, String3
2
3   String1 = "The quick brown fox jumps over the lazy dog."
4
5   ; Left returns a number of characters from the left hand end of a
     string.
6   ; Mid returns a number of characters from the given start location in
     the middle of a string.
7   ; Right returns a number of characters from the right hand end of a
     string.
8   ; Space returns the specified number of space characters as a string.
9   ; Shows "The brown dog."
10  Debug "* Left, Mid and Right"
11  String2 = Left(String1, 3) + Space(1) + Mid(String1, 11, 5) +
     Space(1) + Right(String1, 4)
12  Debug String2
13
14  ; CountString returns the number of instances of the second string in
     the first string, it is case sensitive.
15  ; Shows 1.
16  Debug "* CountString"
17  Debug CountString(String1, "the")
18
19  ; However the LCase (and UCase) functions can be used to switch a
     string to all lower (or upper) case
20  ; Shows 2
21  Debug "* CountString and LCase"
22  String2 = LCase(String1)
23  Debug CountString(String2, "the")
24
25  ; FindString can be used to find the location of one string within
     another.
26  ; Shows 17.
27  Debug "* FindString"
28  Debug FindString(String1, "fox")
29
30  ; RemoveString can be used to remove one string from within another.
31  ; Shows The quick fox jumps over the lazy dog.
32  Debug "* RemoveString"
```

```
33  String2 = RemoveString(String1, " brown")
34  Debug String2
35
36  ; ReplaceString can be used to change the occurrence of a substring
     within another string.
37  ; Shows The quick brown fox jumps over the sleeping dog.
38  Debug "* ReplaceString"
39  String2 = ReplaceString(String1, "lazy", "sleeping")
40  Debug String2
41
42  ; StringByteLength returns the length of a string in bytes in the
     specified format, or the current default
43  ; if one is not specified (excluding the terminating null).
44  Debug "* StringByteLength"
45  ; Shows 44.
46  Debug StringByteLength(String1, #PB_Ascii)
47  ; Shows 88.
48  Debug StringByteLength(String1, #PB_Unicode)
49
50  ; StringField can be used to obtain an indexed substring from a
     target string.
51  ; Useful for converting strings to lists for example.
52  ; StringField will work with space as a delimiter too
53  ; but hopefully this example makes the functions behaviour more
     apparent.
54  ; Shows jumps.
55  Debug "* StringField"
56  String2 = ReplaceString(String1, " ", "\")
57  Debug String2
58  String3 = StringField(String2, 5, "\")
59  Debug String3
60
61  ; Trim removes white space characters from the start and end of a
     given string.
62  ; Similarly, LTrim acts on the left hand end (start) of a string and
     RTrim the right hand end.
63  Debug "* Trim, LTrim and RTrim"
64  String2 = Space(10) + String1 + Space(8)
65  Debug #DQUOTE$ + String2 + #DQUOTE$
66  String3 = Trim(String2)
67  Debug #DQUOTE$ + String3 + #DQUOTE$
68  String3 = LTrim(String2)
69  Debug #DQUOTE$ + String3 + #DQUOTE$
70  String3 = RTrim(String2)
71  Debug #DQUOTE$ + String3 + #DQUOTE$
72
73  ; LSet sets a string to be a specific length from the left hand end,
     padding with spaces,
74  ; or other specified character, as necessary.
75  ; If the string is already longer than the specified length it will
     be truncated.
76  Debug "*LSet"
77  Debug LSet("Abc", 10, "*")
78  Debug LSet("Abcd", 10, "*")
79  Debug LSet("Abcde", 10, "*")
80
81  ; Similarly RSet pads a string from its right hand end.
82  Debug "* RSet"
83  Debug RSet("1.23", 10, "0")
```

```
84    Debug RSet("10.23", 10, "0")
85    Debug RSet("100.23", 10, "0")
86
87    ; Str converts a signed quad value to a string, similarly StrF
       converts floats,
88    ; StrD converts doubles and StrU converts unsigned values, these two
       function have an optional
89    ; parameter to specify the number of decimal places to show.
90    Debug "* Str, StrF and StrD"
91    Debug Str(100)
92    Debug StrF(1.234, 3)
93    Debug StrD(123456.789, 3)
94
95    ; Val will convert a string value into its numeric (quad) equivalent.
96    ; ValD and ValF perform the same function for floats and doubles.
97    Debug "* Val"
98    Debug Val("123")
99
100   ; Bin will convert a numeric value into its binary equivalent.
101   ; Hex will convert one into its hexadecimal equivalent.
102   Debug "* Bin and Hex"
103   Debug Bin(19)
104   Debug Hex(19)
```

UserGuide Navigation

< Previous: Loops || Overview || Next: Storing data in memory >

Chapter 91

UserGuide - Displaying text output (Console)

In the previous topic Input & Output you already saw an overview about the different possibilities to output text to the user, and in the topic Storing Data in Memory , we started to build a small application to display the properties of files in a particular folder to the debug window .
Now we're going to revisit this example to improve the data output section to resolve some issues with using the debug window. Firstly, this window is only available in the PureBasic IDE which means its only useful to programmers, secondly it doesn't really give us much control over how our output looks. PureBasic provides a text mode window, or console window , which can be used in compiled programs. So let's update our example to use it instead.
First, we will need some extra working variables to make this work properly. Amend the variable definitions like this:

```
...

; Now we define a new list of files using the structure previously specified
NewList Files.FILEITEM()
Define.s Access, Attrib, Create, Folder, Modify, Msg, Num, Size
Define.l Result

...
```

Next, remove the output section of code completely, from the comment line:

```
; If there are some entries in the list, show the results in the debug window.
...
```

Now replace this with:

```
; Open a text mode screen to show the results.
OpenConsole()

; Display a title.
; PrintN displays the string given in the console window and moves the print position to the start of the next line afterwards.
; Space(n) returns n spaces in a string.
PrintN("File list of " + Folder + ".")
PrintN("-------------------------------------------------------------------------------
Msg = "Num Name"
PrintN(Msg)
```

```
  Msg = Space(4) + "Create" + Space(5) + "Access" + Space(5) + "Modify"
   + Space(5) + "Attrib Size"
  PrintN(Msg)

  ; Loop through the list to display the results.
  ForEach Files()

    ; Tabulate the number of the list index.
    ; ListIndex() returns the current position in the list, counting
   from zero.
    ; StrU converts an unsigned number into a string.
    ; RSet pads a string to a set length with the necessary number of a
   specified character at the front.
    ; Here we use it to make sure all the index numbers are padded to 3
   characters in length.
    Num = RSet(StrU(ListIndex(Files()) + 1), 3, " ")

    ; Display the item number and file name.
    Msg = Num + " " + Files()\Name
    PrintN(Msg)

    ; These lines convert the three date values to something more
   familiar.
    Create = FormatDate("%dd/%mm/%yyyy", Files()\DateCreated)
    Access = FormatDate("%dd/%mm/%yyyy", Files()\DateAccessed)
    Modify = FormatDate("%dd/%mm/%yyyy", Files()\DateModified)

    ; Convert the file size to a padded string the same as with the
   index value above,
    ; but allow space for the maximum size of a quad.
    Size = RSet(StrU(Files()\Size), 19)

    ; Convert the attributes to a string, for now.
    Attrib = RSet(StrU(Files()\Attributes), 6, " ")

    ; Display the file's properties.
    Msg = Space(4) + Create + " " + Access + " " + Modify + " " +
   Attrib + " " + Size
    PrintN(Msg)

    ; Display a blank line.
    PrintN("")

  Next Files()

  ; Wait for the return key to be displayed, so the results can be
   viewed before the screen closes.
  PrintN("")
  PrintN("Press return to exit")
  Input()
```

All being well the output should appear in a console window looking something like this:

```
  File List of C:\Documents And Settings\user\.
  ------------------------------------------------------------------------------
  Num Name
      Create      Access      Modify      Attrib Size
    1 NTUSER.DAT
      03/07/2008 04/04/2011 02/04/2011     34            18874368
```

```
7
8       2 kunzip.dll
9         14/07/2008  04/04/2011  14/07/2008      32                 18432
10
11      3 ntuser.dat.LOG
12        03/07/2008  04/04/2011  04/04/2011      34                  1024
13
14      4 ntuser.ini
15        03/07/2008  02/04/2011  02/04/2011       6                   278
16
17    Press Return To exit
```

This output is from a Windows XP system, later versions of Windows and Linux or Mac OSX will show different files of course.
Note for Linux/MacOS: Please note, to select "Console" as executable format in the compiler options .

UserGuide Navigation

< Previous: Input & Output || Overview || Next: Building a graphical user interface (GUI) >

Chapter 92

UserGuide - Some Tips & Tricks

"Using a map to index a list"

lists provide a powerful way to build a structured storage system - however they have a disadvantage. If you aren't sure exactly where in the list a particular item is, you must examine each entry in the list to find the right one.
Maps also provide a similar function but are indexed by a key value instead - however they too have a disadvantage, they don't maintain the order elements are inserted into the list.
However by using a combination of the two, you can neatly avoid both of these issues...
This example loads a structured list of book data and builds an index of ISBN numbers using a map . It then demonstrates how to access the list using the index in the map.

```
1   EnableExplicit
2
3   ; A book catalog structure.
4   Structure BOOK
5     Title.s
6     Author.s
7     ISBN13.s
8     Price.d
9   EndStructure
10
11  ; Create a list to hold the catalog entries.
12  NewList Catalog.BOOK()
13  ; Create a map to hold the ISBN index.
14  NewMap ISBN13.l()
15  ; Working variables.
16  Define.l Count, Index
17  Define.s ISBN
18
19  ; Add an empty element at the top of the list.
20  ; The first element in a list is numbered zero, however the map will
      return zero if a requested entry isn't present.
21  ; This empty element avoids a potential problem with an incorrect
      reference to the first catalog item being returned.
22  AddElement(Catalog())
23
24  For Count = 1 To 5
25
26    ; Read the data from the table into the list.
27    AddElement(Catalog())
28    Read.s Catalog()\Title
29    Read.s Catalog()\Author
```

```
    Read.s Catalog()\ISBN13
    Read.d Catalog()\Price

    ; Index the ISBN to the map.
    ISBN13(Catalog()\ISBN13) = ListIndex(Catalog())

  Next Count

  ; Find an entry.
  ISBN = "978-0340896983"
  Index = ISBN13(ISBN)

  If Index > 0
    Debug "Book with ISBN13 " + ISBN + " is at list index " +
   StrU(Index) + "."
    Debug ""

    ; We can now directly select the right list element without having
   to perform a search.
    SelectElement(Catalog(), Index)
    Debug "'" + Catalog()\Title + "' by " + Catalog()\Author
    Debug "ISBN: " + Catalog()\ISBN13
    Debug "Price: " + StrD(Catalog()\Price, 2)

  Else
    Debug "No book with that ISBN in the catalog."

  EndIf

  End

  ; Some test data.
  DataSection

    BookData:

    Data.s "Carte Blanche", "Jeffery Deaver", "978-1444716474"
    Data.d 19.99

    Data.s "One Day", "David Nicholls", "978-0340896983"
    Data.d 7.99

    Data.s "Madeleine", "Kate McCann", "978-0593067918"
    Data.d 20.00

    Data.s "The Dukan Diet", "Dr Pierre Dukan", "978-1444710335"
    Data.d 8.99

    Data.s "A Game of Thrones", "George R. R. Martin", "978-0006479888"
    Data.d 9.99

    Data.s "The Help", "Kathryn Stockett", "978-0141039282"
    Data.d 8.99

  EndDataSection
```

UserGuide Navigation

< Previous: Advanced functions || Overview

Chapter 93

UserGuide - Variables and Processing of variables

A is an integer - but note that you can't declare variables this way if you use the EnableExplicit directive. Outside of a procedure A will exist at Main scope, within a procedure it will become local.

```
1  A = 0
```

B is a long integer, **C** is a floating-point number, they will be initialized to zero.

```
1  Define B.l, C.f
```

D and **E** are long integers too. However, they will be initialized to the stated constants. Notice too the alternative syntax of Define used.

```
1  Define.l D = 10, E = 20
```

F is a string.

```
1  Define.s F
```

So is **G$**, however, if you declare strings this way, you must always use the $ notation.

```
1  G$ = "Hello, "
```

This won't work. (G becomes a new integer variable and a compiler error occurs).

```
1  G = "Goodbye, World!"
```

H is an array of 20 strings, array indexing begins at zero.

```
1  Dim H.s(19)
```

Now H is an array of 25 strings. If the array is resized larger, its original contents will be preserved.

```
1  ReDim H.s(24)
```

J will appear at Global , rather than Main, scope. It will appear in all procedures, but maybe a better way would be to use the Shared keyword inside a procedure on a main scope variable, so that the chances of accidental changes are minimized.

```
1  Global.i J
```

K will be a new, empty, list of strings at main scope.

```
NewList K.s()
```

M will be a new, empty, map of strings at main scope.

```
NewMap M.s()
```

Note that you can't use the alternative syntax of the Define keyword with NewList or NewMap though. A compiler error will result.
Within a procedure:

```
Procedure TestVariables()
??
?? ; N and P will be local.
?? Define.l N
?? Protected.l P
??
?? ; Q will be a static local.
?? Static.l Q
??
?? ; The main scope variable F and the string list K will be
 available within this procedure.
?? Shared F, K()
??
?? ; The global scope variable J will be available here implicitly.
??
EndProcedure
```

Using operators on variables:

```
; Add two to A.
A + 2

; Bitwise Or with 21 (A will become 23)
A | 21

; Bitwise And with 1 (A will become 1)
A & 1

; Arithmetic shift left (A will become 2, that is 10 in binary).
A << 1
```

String concatenation:

```
G$ + "World!"
```

Add an element to the K list:

```
AddElement(K())
K() = "List element one"
```

Add an element to the M map:

```
M("one") = "Map element one"
```

UserGuide Navigation

< Previous: First steps || Overview || Next: Constants >

Chapter 94

Unicode

Introduction

Unicode is the term used for extended character sets which allows displaying text in many languages (including those with a lot of different characters like Japanese, Korean etc.). It solves the ASCII problem of having a lot of table dedicated for each language by having a unified table where each character has its own place.
To simplify, unicode can be seen as a big ascii table, which doesn't have 256 characters but up to 65536. Thus, each character in unicode will need 2 bytes in memory (this is important when dealing with pointers for example).
Here are some links to have a better understanding about unicode (must have reading):
General unicode information
Unicode and BOM

Unicode and Windows

PureBasic internally uses the UCS2 encoding which is also the format used by the Windows unicode API, so no conversions are needed at runtime when calling an OS function. When dealing with an API function, PureBasic will automatically use the unicode one if available (for example MessageBox_() will map to MessageBoxW()). All the API structures and constants supported by PureBasic are using unicode version.

UTF-8

UTF-8 is another unicode encoding, which is byte based. Unlike UCS2 which always takes 2 bytes per characters, UTF-8 uses a variable length encoding for each character (up to 4 bytes can represent one character). The biggest advantage of UTF-8 is the fact it doesn't includes null characters in its coding, so it can be edited like a regular text file. Also the ASCII characters from 1 to 127 are always preserved, so the text is kind of readable as only special characters will be encoded. One drawback is its variable length, so it all string operations will be slower due to needed pre-processing to locate a character is in the text.
PureBasic uses UTF-8 by default when writing string to files (File and Preference libraries), so all texts are fully cross-platform.
The PureBasic compiler also handles both Ascii and UTF-8 files (the UTF-8 files need to have the correct BOM header to be handled correctly). Both can be mixed in a single program without problem: an ascii file can include an UTF-8 file and vice-versa, as long as all string-literals (i.e. "x") and character-literals (i.e. 'x') in the ASCII files contain only characters with a code <= 127.. When developing, it's recommended to set the IDE in UTF-8 mode, so all the source files will be unicode ready.

Chapter 95

Variables and Types

Variables declaration

To declare a variable in PureBasic, simply type its name. You can also specify the type you want this variable to be. By default, when a data type is not indicated, the data is an integer. Variables do not need to be explicitly declared, as they can be used as "variables on-the-fly". The Define keyword can be used to declare multiple variables in one statement. If you don't assign an initial value to the variable, their value will be 0.

Example

```
1  a.b          ; Declare a variable called 'a' from byte (.b) type.
2  c.l = a*d.w ; 'd' is declared here within the expression and it's a
     word !
```

Notes:

Variable names must not start with a number (0,1,...), contain operators (+,-,...) or special characters (ß,ä,ö,ü,...).
The variables in PureBasic are not case sensitive, so "pure" and "PURE" are the same variable.
If you don't need to change the content of a variable during the program flow (e.g. you're using fixed values for ID's etc.), you can also take a look at constants as an alternative.
To avoid typing errors etc. it's possible to force the PureBasic compiler to always want a declaration of variables, before they are first used. Just use EnableExplicit keyword in your source code to enable this feature.

Basic types

PureBasic allows many type variables which can be standard integers, float, double, quad and char numbers or even string characters. Here is the list of the native supported types and a brief description :

```
Name          | Extension  | Memory consumption                | Range
--------------+------------+-----------------------------------+--------------------
Byte          |    .b      | 1 byte                            | -128 to
 +127
Ascii         |    .a      | 1 byte                            | 0 to
 +255
Character     |    .c      | 2 bytes                           | 0 to
 +65535
```

```
Word          |    .w      | 2 bytes                              | -32768
 to +32767
Unicode       |    .u      | 2 bytes                              | 0 to
 +65535
Long          |    .l      | 4 bytes                              |
 -2147483648 to +2147483647
Integer       |    .i      | 4 bytes (using 32-bit compiler)      |
 -2147483648 to +2147483647
Integer       |    .i      | 8 bytes (using 64-bit compiler)      |
 -9223372036854775808 to +9223372036854775807
Float         |    .f      | 4 bytes                              |
 unlimited (see below)
Quad          |    .q      | 8 bytes                              |
 -9223372036854775808 to +9223372036854775807
Double        |    .d      | 8 bytes                              |
 unlimited (see below)
String        |    .s      | string length + 1                    |
 unlimited
Fixed String | .s{Length} | string length                        |
 unlimited
```

Unsigned variables: Purebasic offers native support for unsigned variables with byte and word types via the ascii (.a) and unicode (.u) types. The character (.c) type is an unsigned word in unicode that may be used as an unsigned type.
Notation of string variables: it is possible to use the '$' as last char of a variable name to mark it as string. This way you can use 'a$' and 'a.s' as different string variables. Please note, that the '$' belongs to the variable name and must be always attached, unlike the '.s' which is only needed when the string variable is declared the first time.

```
1  a.s = "One string"
2  a$ = "Another string"
3  Debug a    ; will give "One string"
4  Debug a$   ; will give "Another string"
```

Note: The floating numbers (floats + doubles) can also be written like this: 123.5e-20

```
1  value.d = 123.5e-20
2  Debug value    ; will give 0.00000000000000001235
```

Operators

Operators are the functions you can use in expressions to combine the variables, constants, and whatever else. The table below shows the operators you can use in PureBasic, in no particular order (LHS = Left Hand Side, RHS = Right Hand Side).

Operator = (Equals)

This can be used in two ways. The first is to assign the value of the expression on the RHS to the variable on the LHS. The second way is when the result of the operator is used in an expression and is to test whether the values of the expressions on the LHS and RHS are the same (if they are the same this operator will return a true result, otherwise it will be false).

Example

```
a = b+c       ; Assign the value of the expression "b+c" to the
 variable "a"
If abc = def ; Test if the values of abc and def are the same, and
 use this result in the If command
```

Operator + (Plus)

Gives a result of the value of the expression on the RHS added to the value of the expression on the LHS. If the result of this operator is not used and there is a variable on the LHS, then the value of the expression on the RHS will be added directly to the variable on the LHS.

Example

```
number=something+2 ; Adds the value 2 to "something" and uses the
 result with the equals operator
variable+expression ; The value of "expression" will be added
 directly to the variable "variable"
```

Operator - (Minus)

Subtracts the value of the expression on the RHS from the value of the expression on the LHS. When there is no expression on the LHS this operator gives the negative value of the value of the expression on the RHS. If the result of this operator is not used and there is a variable on the LHS, then the value of the expression on the RHS will be subtracted directly from the variable on the LHS. This operator cannot be used with string type variables.

Example

```
var=#MyConstant-foo ; Subtracts the value of "foo" from "#MyConstant"
 and uses the result with the equals operator
another=another+ -var ; Calculates the negative value of "var" and
 uses the result in the plus operator
variable-expression ; The value of "expression" will be subtracted
 directly from the variable "variable"
```

Operator * (Multiplication)

Multiplies the value of the expression on the LHS by the value of the expression on the RHS. If the result of this operator is not used and there is a variable on the LHS, then the value of the variable is directly multiplied by the value of the expression on the RHS. This operator cannot be used with string type variables.

Example

```
total=price*count ; Multiplies the value of "price" by the value of
 "count" and uses the result with the equals operator
variable*expression ; "variable" will be multiplied directly by the
 value of "expression"
```

Operator / (Division)

Divides the value of the expression on the LHS by the value of the expression on the RHS. If the result of this operator is not used and there is a variable on the LHS, then the value of the variable is directly divided by the value of the expression on the RHS. This operator cannot be used with string type variables.

Example

```
1 count=total/price ; Divides the value of "total" by the value of
   "price" and uses the result with the equals operator
2 variable/expression ; "variable" will be divided directly by the
   value of "expression"
```

Operator & (Bitwise AND)

You should be familiar with binary numbers when using this operator. The result of this operator will be the value of the expression on the LHS anded with the value of the expression on the RHS, bit for bit. The value of each bit is set according to the table below. Additionally, if the result of the operator is not used and there is a variable on the LHS, then the result will be stored directly in that variable. This operator cannot be used with strings.

```
LHS | RHS | Result
------------------
 0  |  0  |    0
 0  |  1  |    0
 1  |  0  |    0
 1  |  1  |    1
```

Example

```
1 ; Shown using binary numbers as it will be easier to see the result
2 a.w = %1000 & %0101 ; Result will be 0
3 b.w = %1100 & %1010 ; Result will be %1000
4 bits = a & b ; AND each bit of a and b and use result in equals
   operator
5 a & b ; AND each bit of a and b and store result directly in variable
   "a"
```

Operator || (Bitwise OR)

You should be familiar with binary numbers when using this operator. The result of this operator will be the value of the expression on the LHS or'ed with the value of the expression on the RHS, bit for bit. The value of each bit is set according to the table below. Additionally, if the result of the operator is not used and there is a variable on the LHS, then the result will be stored directly in that variable. This operator cannot be used with strings.

```
LHS | RHS | Result
------------------
 0  |  0  |    0
 0  |  1  |    1
 1  |  0  |    1
 1  |  1  |    1
```

Example

```
1  ; Shown using binary numbers as it will be easier to see the result
2  a.w = %1000 | %0101 ; Result will be %1101
3  b.w = %1100 | %1010 ; Result will be %1110
4  bits = a | b ; OR each bit of a and b and use result in equals
     operator
5  a | b ; OR each bit of a and b and store result directly in variable
     "a"
```

Operator ! (Bitwise XOR)

You should be familiar with binary numbers when using this operator. The result of this operator will be the value of the expression on the LHS xor'ed with the value of the expression on the RHS, bit for bit. The value of each bit is set according to the table below. Additionally, if the result of the operator is not used and there is a variable on the LHS, then the result will be stored directly in that variable. This operator cannot be used with strings.

```
LHS | RHS | Result
------------------
 0  |  0  |    0
 0  |  1  |    1
 1  |  0  |    1
 1  |  1  |    0
```

Example

```
1  ; Shown using binary numbers as it will be easier to see the result
2  a.w = %1000 ! %0101 ; Result will be %1101
3  b.w = %1100 ! %1010 ; Result will be %0110
4  bits = a ! b ; XOR each bit of a and b and use result in equals
     operator
5  a ! b ; XOR each bit of a and b and store result directly in variable
     "a"
```

Operator * * (Bitwise NOT)

You should be familiar with binary numbers when using this operator. The result of this operator will be the not'ed value of the expression on the RHS, bit for bit. The value of each bit is set according to the table below. This operator cannot be used with strings.

```
RHS | Result
----------
 0  |    1
 1  |    0
```

Example

```
1  ; Shown using binary numbers as it will be easier to see the result
2  a.w = ~%1000 ; Result will be %0111
3  b.w = ~%1010 ; Result will be %0101
```

Operator () (Parentheses)

You can use sets of parentheses to force part of an expression to be evaluated first, or in a certain order.

Example

```
1  a = (5 + 6) * 3 ; Result will be 33 since the 5+6 is evaluated first
2  b = 4 * (2 - (3 - 4)) ; Result will be 12 since the 3-4 is evaluated
     first, then the 2-result, then the multiplication
```

Operator < (Less than)

This is used to compare the values of the expressions on the LHS and RHS. If the value of the expression on the LHS is less than the value of the expression on the RHS this operator will give a result of true, otherwise the result is false.

Operator > (More than)

This is used to compare the values of the expressions on the LHS and RHS. If the value of the expression on the LHS is more than the value of the expression on the RHS this operator will give a result of true, otherwise the result is false.

Operator <=, =< (Less than or equal to)

This is used to compare the values of the expressions on the LHS and RHS. If the value of the expression on the LHS is less than or equal to the value of the expression on the RHS this operator will give a result of true, otherwise the result is false.

Operator >=, => (More than or equal to)

This is used to compare the values of the expressions on the LHS and RHS. If the value of the expression on the LHS is more than or equal to the value of the expression on the RHS this operator will give a result of true, otherwise the result is false.

Operator <> (Not equal to)

This is used to compare the values of the expressions on the LHS and RHS. If the value of the expression on the LHS is equal to the value of the expression on the RHS this operator will give a result of false, otherwise the result is true.

Operator And (Logical AND)

Can be used to combine the logical true and false results of the comparison operators to give a result shown in the following table.

```
  LHS   |   RHS   | Result
------------------------
 false | false | false
```

```
 false |  true | false
  true | false | false
  true |  true |  true
```

Operator Or (Logical OR)

Can be used to combine the logical true and false results of the comparison operators to give a result shown in the following table.

```
  LHS  |  RHS  | Result
-----------------------
 false | false | false
 false |  true |  true
  true | false |  true
  true |  true |  true
```

Operator XOr (Logical XOR)

Can be used to combine the logical true and false results of the comparison operators to give a result shown in the following table. This operator cannot be used with strings.

```
  LHS  |  RHS  | Result
-----------------------
 false | false | false
 false |  true |  true
  true | false |  true
  true |  true | false
```

Operator Not (Logical NOT)

The result of this operator will be the not'ed value of the logical on the RHS. The value is set according to the table below. This operator cannot be used with strings.

```
  RHS  | Result
---------------
 false |  true
  true | false
```

Operator « (Arithmetic shift left)

Shifts each bit in the value of the expression on the LHS left by the number of places given by the value of the expression on the RHS. Additionally, when the result of this operator is not used and the LHS contains a variable, that variable will have its value shifted. It probably helps if you understand binary numbers when you use this operator, although you can use it as if each position you shift by is multiplying by an extra factor of 2.

Example

```
1  a=%1011 << 1 ; The value of a will be %10110. %1011=11, %10110=22
2  b=%111 << 4 ; The value of b will be %1110000. %111=7, %1110000=112
3  c.l=$80000000 << 1 ; The value of c will be 0. Bits that are shifted
     off the left edge of the result are lost.
```

Operator » (Arithmetic shift right)

Shifts each bit in the value of the expression on the LHS right by the number of places given by the value of the expression on the RHS. Additionally, when the result of this operator is not used and the LHS contains a variable, that variable will have its value shifted. It probably helps if you understand binary numbers when you use this operator, although you can use it as if each position you shift by is dividing by an extra factor of 2.

Example

```
1  d=16 >> 1 ; The value of d will be 8. 16=%10000, 8=%1000
2  e.w=%10101010 >> 4 ; The value of e will be %1010. %10101010=170,
     %1010=10. Bits shifted out of the right edge of the result are lost
     (which is why you do not see an equal division by 16)
3  f.b=-128 >> 1 ; The value of f will be -64. -128=%10000000,
     -64=%11000000. When shifting to the right, the most significant bit
     is kept as it is.
```

Operator % (Modulo)

Returns the remainder of the RHS by LHS integer division.

Example

```
1  a=16 % 2 ; The value of a will be 0 as 16/2 = 8 (no remainder)
2  b=17 % 2 ; The value of a will be 1 as 17/2 = 8*2+1 (1 is remaining)
```

Operators shorthands

Every math operators can be used in a shorthand form.

Example

```
1  Value + 1  ; The same as: Value = Value + 1
2  Value * 2  ; The same as: Value = Value * 2
3  Value << 1 ; The same as: Value = Value << 1
```

Note: this can lead to 'unexpected' results is some rare cases, if the assignment is modified before the affection.

Example

```
1  Dim MyArray(10)
2  MyArray(Random(10)) + 1 ; The same as: MyArray(Random(10)) =
     MyArray(Random(10)) + 1, but here Random() won't return the same
     value for each call.
```

Operators priorities

```
Priority Level |      Operators
---------------+---------------------------
      8 (high)  |          ~, - (negative)
      7         |       <<, >>, %, !
      6         |          |, &
      5         |          *, /
      4         |          +, - (subtraction)
      3         | >, >=, =>, <, <=, =<, =, <>
      2         |           Not
      1 (low)   |       And, Or, XOr
```

Structured types

Build structured types, via the Structure keyword. More information can be found in the structures chapter .

Pointer types

Pointers are declared with a '*' in front of the variable name. More information can be found in the pointers chapter .

Special information about Floats and Doubles

A floating-point number is stored in a way that makes the binary point "float" around the number, so that it is possible to store very large numbers or very small numbers. However, you cannot store very large numbers with very high accuracy (big and small numbers at the same time, so to speak).
Another limitation of floating-point numbers is that they still work in binary, so they can only store numbers exactly which can be made up of multiples and divisions of 2. This is especially important to realize when you try to print a floating-point number in a human readable form (or when performing operations on that float) - storing numbers like 0.5 or 0.125 is easy because they are divisions of 2. Storing numbers such as 0.11 are more difficult and may be stored as a number such as 0.10999999. You can try to display to only a limited range of digits, but do not be surprised if the number displays different from what you would expect!
This applies to floating-point numbers in general, not just those in PureBasic.
Like the name says the doubles have double-precision (64-bit) compared to the single-precision of the floats (32-bit). So if you need more accurate results with floating-point numbers use doubles instead of floats.
The exact range of values, which can be used with floats and doubles to get correct results from arithmetic operations, looks as follows:

Float: +- 1.175494e-38 till +- 3.402823e+38
Double: +- 2.2250738585072013e-308 till +- 1.7976931348623157e+308

More information about the 'IEEE 754' standard you can get on Wikipedia.

Chapter 96

While : Wend

Syntax

```
While <expression>
  ...
Wend
```

Description

Wend will loop until the <expression> becomes false. A good point to keep in mind with a While test is that if the first test is false, then the program will never enter the loop and will skip this part. A Repeat loop is executed at least once, (as the test is performed after each loop).
With the Break command it is possible to exit the While : Wend loop during any iteration, with the Continue command the end of the current iteration may be skipped.

Example

```
1  b = 0
2  a = 10
3  While a = 10
4    b = b+1
5    If b=10
6      a=11
7    EndIf
8  Wend
```

This program loops until the 'a' value is <> 10. The value of 'a' becomes 11 when b=10, the program will loop 10 times.

Chapter 97

Windows Message Handling

Introduction

The messages for your program will be sent by Windows into a queue, which is worked off only if you want this. Windows sends thousand messages to your program without noticing this directly.
For example if you change the status of a gadget (identical whether you add a new entry or change the image of an ImageGadget), a message is sent to the queue of your program.
There are two possibilities to receive and to process the Windows messages in PureBasic: WaitWindowEvent() and WindowEvent() . The difference is, that WaitWindowEvent() waits till a message arrives and WindowEvent() continues to work also so. The messages in the queue will be processed however only, after you have called WindowEvent() or WaitWindowEvent().

Specials of WindowEvent()

The command WindowEvent() don't wait, until a message arrives, but checks only whether one is in the queue. If yes, the message is processed and WindowEvent() gives back the number of the message. If no message lines up, then zero (0) is given back.

Example

```
1  While WindowEvent() : Wend
```

cause, that WindowEvent() becomes called as long till it gives back 0, i.e. until all messages of the queue are processed.
It doesn't reach, if you insert a simple 'WindowEvent()' after a SetGadgetState() to process this 1 message. Firstly there still can be other messages in the queue, which have arrived before, and secondly Windows also sends quite a number of other messages, we don't have to take care of... which nevertheless are in the queue.
A simple call of

```
1  WindowEvent()
```

doesn't reach, the code then runs correct under circumstances on one Windows version, but on another version not. The different Windows versions are internally very different, so that one version only sends 1 message but another Windows version sends 5 messages for the same circumstance.
Because of this one always uses for updating:

```
1  While WindowEvent() : Wend
```

Of course there is also the alternative

```
1   Repeat : Until WindowEvent() = 0
```

possible, what isn't rather usual however.
The described method While : WindowEvent() : Wend is frequently useful in connection with the command Delay() , where the loop is inserted BEFORE the Delay(), e.g. to firstly wait for the update of an ImageGadget after changing an image with SetGadgetState().

Chapter 98

With : EndWith

Syntax

```
With <expression>
  ...
EndWith
```

Description

With : EndWith blocks may be used with structure fields in order to reduce the quantity of code and to improve its' readability. This is a compiler directive, and works similarly to a macro , i.e., the specified expression is automatically inserted before any backslash '\' character which have a space or an operator preceding it. The code behaves identically to its' expanded version. With : EndWith blocks may not be nested, as this could introduce bugs which are difficult to track under conditions where several statements have been replaced implicitly.

Example

```
1   Structure Person
2     Name$
3     Age.l
4     Size.l
5   EndStructure
6
7   Friend.Person
8
9   With Friend
10    \Name$ = "Yann"
11    \Age   = 30
12    \Size  = 196
13
14    Debug \Size+\Size
15  EndWith
```

Example: Complex example

```
1   Structure Body
2     Weight.l
3     Color.l
```

```
    Texture.l
  EndStructure

  Structure Person
    Name$
    Age.l
    Body.Body[10]
  EndStructure

  Friend.Person

  For k = 0 To 9
    With Friend\Body[k]
      \Weight = 50
      \Color  = 30
      \Texture = \Color*k

      Debug \Texture
    EndWith
  Next
```