SpiderBasic Reference Manual
3.02

http://www.spiderbasic.com/

February 12, 2025

Contents

Part I

General

Chapter 1

Introduction

SpiderBasic is an ”high-level” programming language based on established ”BASIC” rules. It does share background with other ”BASIC” compiler, but has its own syntax extensions. Learning SpiderBasic is very easy! SpiderBasic has been created for beginners and experts alike. Compilation time is extremely fast. 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 structures, procedures, dynamic lists and much more. For the experienced coder, there are no problems gaining access to external third party libraries.

The main features of SpiderBasic

- Huge set of internal commands (500+) to quickly and easily build applications or games
- BASIC based keywords
- Very fast compiler which creates optimized apps
- Procedure and structure support for advanced programming
- Built-in containers like array, list and map
- Strong types, strong syntax to avoid programming mistakes
- Full unicode support
- Namespace support for easy code reuse
- Easy but very fast 2D game support through WebGL
- Inlined JavaScript support for extensibility
- Dedicated editor and development environment
- Available on Windows, MacOS X and Linux
- Very close to PureBasic, which allow to port easily an application to the desktop

Chapter 2

Terms And Conditions

This program is provided "**AS IS**". Fantaisie Software are NOT responsible for any damage (or damages) attributed to SpiderBasic. You are warned that you use SpiderBasic 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.
SpiderBasic 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.
Fantaisie Software reserves all rights to this program and all original archives and contents.

Chapter 3

System requirements

SpiderBasic will run on Windows XP, Windows Vista, Windows 7 and Windows 8 (in both 32-bit and 64-bit edition), Linux (kernel 2.2 or above) and MacOS X (10.6 or above).
If there are any problems, please contact us.

Chapter 4

Installation

To install SpiderBasic, just click on the install wizard, follow the steps, and then click on the SpiderBasic icon (found on the desktop or in the start-menu) to launch SpiderBasic.
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 "SpiderBasic\Compilers\" directory to the PATH environment variable to make the compiler accessible from any directory.
Important note: to avoid conflicts with existing SpiderBasic installations (maybe even with user-libraries), please install a new SpiderBasic version always in its own new folder. See also the chapter Using several SpiderBasic versions .

Chapter 5

Order

SpiderBasic is a low-cost programming language. In buying SpiderBasic you will ensure that development will go further and faster. The updates are free until the next major version. For example, when you buy SpiderBasic 1.00, all further 1.xx updates will be free, and the 2.00 version and above will need a new registration fee. For ease of ordering, you can safely use our secure online method. Thanks a lot for your support!

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

- maximum number of source lines: about 800

Full version of SpiderBasic:

Check http://www.spiderbasic.com for more information about pricing.

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:

```
 Andre Beer
 Siedlung 6
 09548 Deutschneudorf
 Germany
 e-mail: andre@spiderbasic.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 Andre.)
```

Delivering of the full version

The full version will be provided via your personal download account, which you will get on www.spiderbasic.com after successful registration. If you order from Andre, 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 contact us, to any of the following addresses:

Frederic 'AlphaSND' Laboureur

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

s-mail :

Frederic Laboureur
10, rue de Lausanne
67640 Fegersheim
France

e-mail : fred@spiderbasic.com

Andre Beer

Andre is responsible for the complete German translation of the SpiderBasic manual and website. SpiderBasic 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@spiderbasic.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. SpiderBasic wouldn't be the same without him !
- **Gaetan Dupont-Panon**: For the wonderful new visual designer, which really rocks on Windows, Linux and OS X !

Miscellaneous

- **Andre Beer**: To spend time for improving the guides (including beginners guide) and do the complete translation into German. Big thanks!

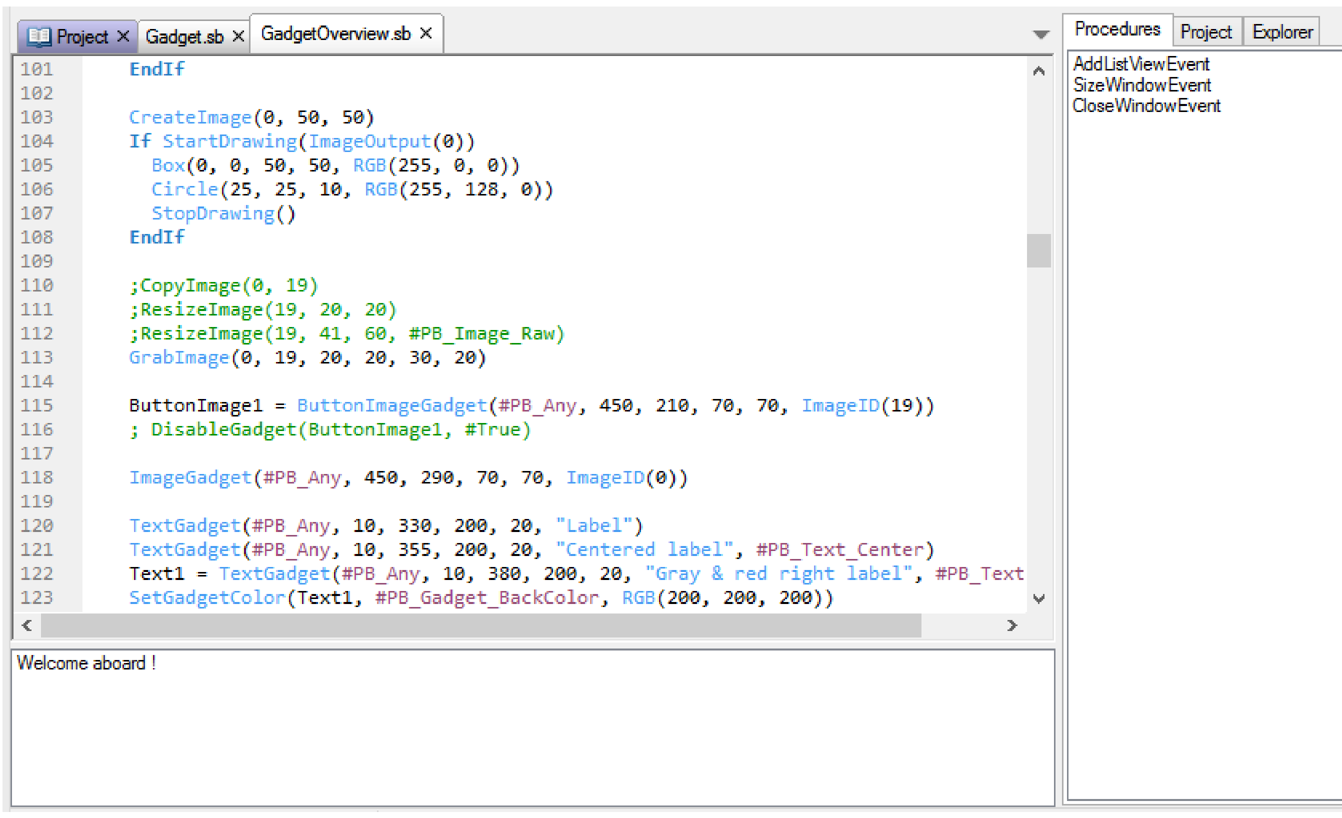
Part II

The SpiderBasic Editor

Chapter 8

Getting Started

The SpiderBasic IDE allows you to create and edit your SpiderBasic source codes, as well as run them, debug them and create the final project.
The IDE main window contains of 3 major parts:



```
    EndIf

    CreateImage(0, 50, 50)
    If StartDrawing(ImageOutput(0))
      Box(0, 0, 50, 50, RGB(255, 0, 0))
      Circle(25, 25, 10, RGB(255, 128, 0))
      StopDrawing()
    EndIf

    ;CopyImage(0, 19)
    ;ResizeImage(19, 20, 20)
    ;ResizeImage(19, 41, 60, #PB_Image_Raw)
    GrabImage(0, 19, 20, 20, 30, 20)

    ButtonImage1 = ButtonImageGadget(#PB_Any, 450, 210, 70, 70, ImageID(19))
    ; DisableGadget(ButtonImage1, #True)

    ImageGadget(#PB_Any, 450, 290, 70, 70, ImageID(0))

    TextGadget(#PB_Any, 10, 330, 200, 20, "Label")
    TextGadget(#PB_Any, 10, 355, 200, 20, "Centered label", #PB_Text_Center)
    Text1 = TextGadget(#PB_Any, 10, 380, 200, 20, "Gray & red right label", #PB_Text
    SetGadgetColor(Text1, #PB_Gadget_BackColor, RGB(200, 200, 200))
```

```
AddListViewEvent
SizeWindowEvent
CloseWindowEvent
```

```
Welcome aboard !
```

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.

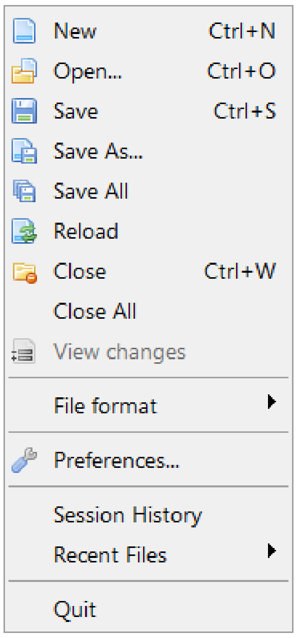
Other then that, there is the main menu and the toolbar. The toolbar simply provides shortcuts to menu features. It can be fully customized. To find out what each button does, move your mouse over it and wait until a small tool-tip appears. It shows the corresponding menu command. The menu commands are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

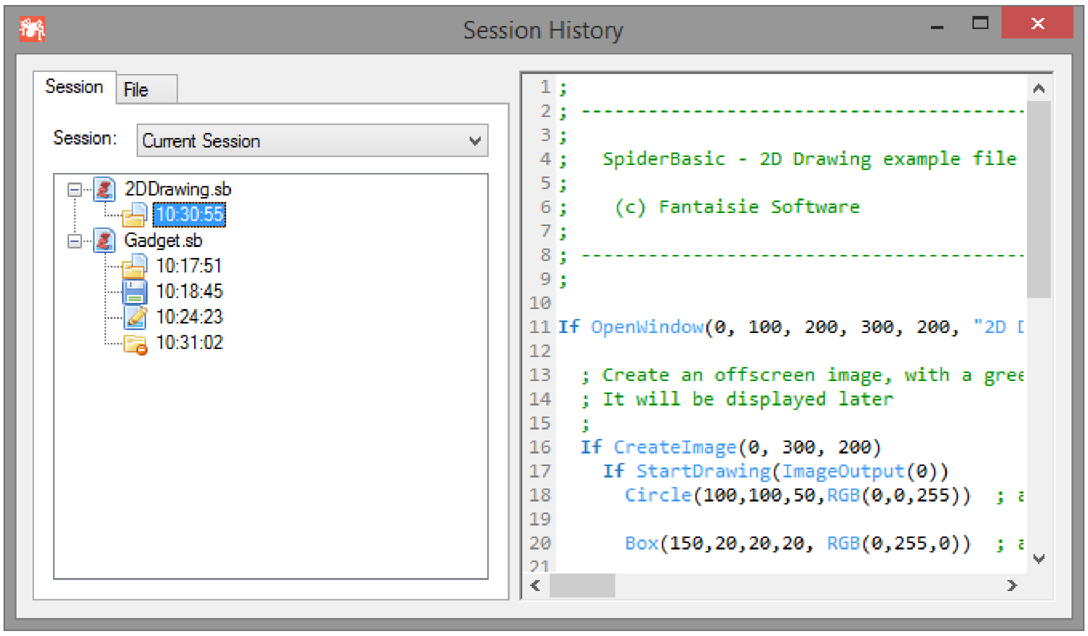
Close All
Closes all currently opened sources.

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 .



```
;
; ----------------------------------------
;
;   SpiderBasic - 2D Drawing example file
;
;    (c) Fantaisie Software
;
; ----------------------------------------
;

If OpenWindow(0, 100, 200, 300, 200, "2D D

  ; Create an offscreen image, with a gree
  ; It will be displayed later
  ;
  If CreateImage(0, 300, 200)
    If StartDrawing(ImageOutput(0))
      Circle(100,100,50,RGB(0,0,255))  ; a

      Box(150,20,20,20, RGB(0,255,0))  ; a
```

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 SpiderBasic 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.
Furthermore, the IDE has many extra editing features specific to programming or SpiderBasic.

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 .

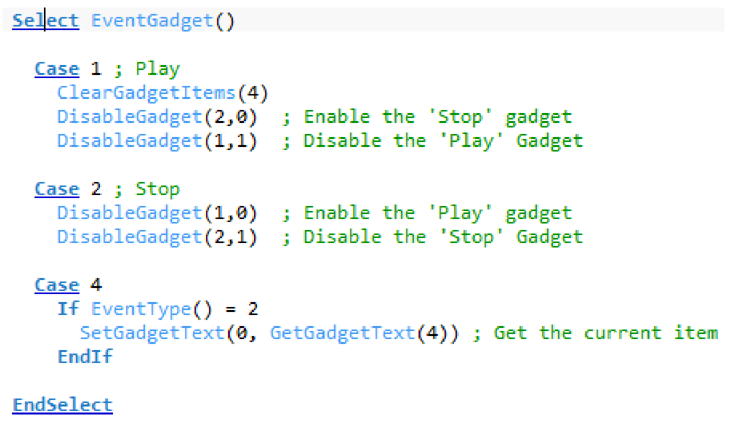
Selecting blocks of code:

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:

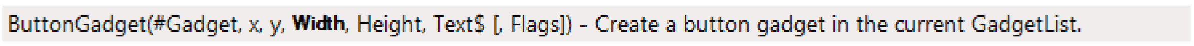
Double-clicking on a word selects the whole word as usual. However in some cases, double-clicking has a special meaning:
When double-clicking on the name of a procedure that is defined in the current source while holding down the Ctrl Key, the cursor automatically jumps to the declaration of this procedure.
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:


```
Select EventGadget()

  Case 1 ; Play
    ClearGadgetItems(4)
    DisableGadget(2,0)  ; Enable the 'Stop' gadget
    DisableGadget(1,1)  ; Disable the 'Play' Gadget

  Case 2 ; Stop
    DisableGadget(1,0)  ; Enable the 'Play' gadget
    DisableGadget(2,1)  ; Disable the 'Stop' Gadget

  Case 4
    If EventType() = 2
      SetGadgetText(0, GetGadgetText(4)) ; Get the current item
    EndIf

EndSelect
```

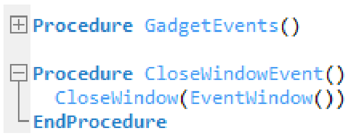
When the cursor is on an opening or closing brace the IDE will highlight the other brace that matches it. If a matching brace could not be found (which is a syntax error in SpiderBasic) the IDE will highlight the current brace in red. This same concept is applied to keywords. If the cursor is on a Keyword 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:


ButtonGadget(#Gadget, x, y, **Width**, Height, Text$ [, Flags]) - Create a button gadget in the current GadgetList.

While typing, the IDE will show the needed parameters for any SpiderBasic 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 or interfaces in your code as long as they are declared in the same source code or project .

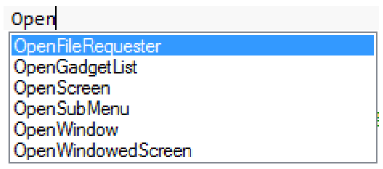
Folding options:



```
⊞Procedure GadgetEvents()

⊟Procedure CloseWindowEvent()
   CloseWindow(EventWindow())
 EndProcedure
```

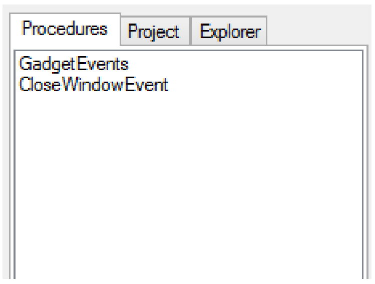
When special folding keywords are encountered (Procedure / EndProcedure by default. More can be added), the IDE marks the region between these keywords on the left side next to the line numbers with a [-] at the starting point, followed by a vertical line to the end point.
By clicking on the [-], you 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:

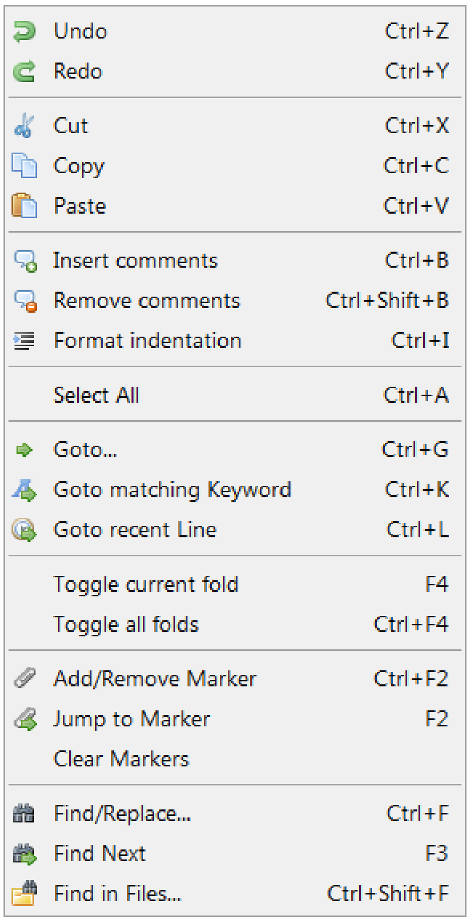


GadgetEvents
CloseWindowEvent

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 .

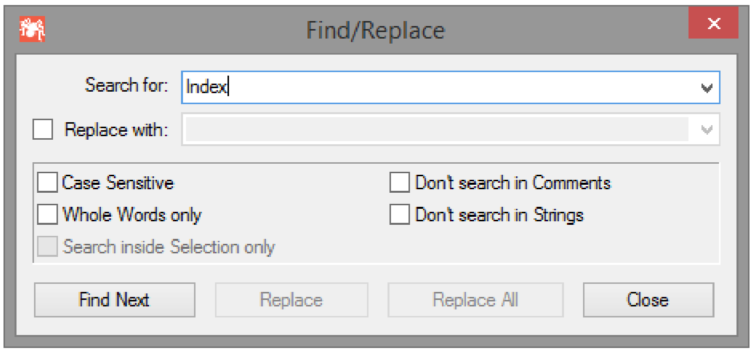
The Edit Menu:

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



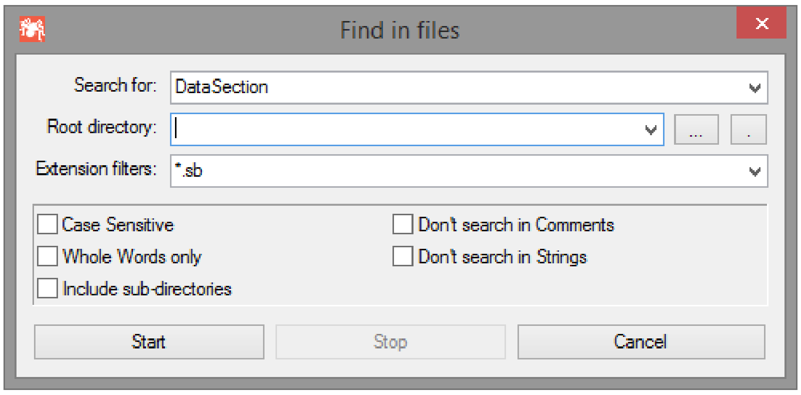
The find/replace dialog enables you to search for specific words in your code, and also to replace them with something else.
The "Find Next" button starts the search. The search can be continued after a match is found with the Find Next menu command (F3 by default).
You can make the search more specific 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 in Files



Search for: DataSection
Root directory:
Extension filters: *.sb
☐ Case Sensitive
☐ Whole Words only
☐ Include sub-directories
☐ Don't search in Comments
☐ Don't search in Strings

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

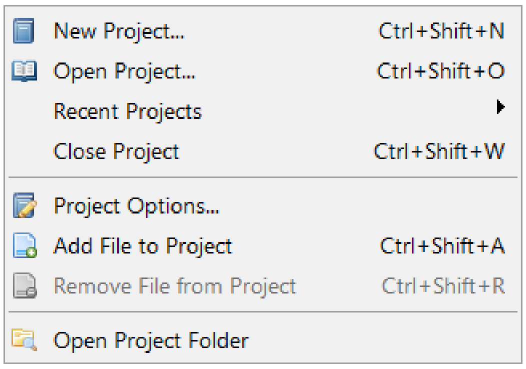
Managing projects

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

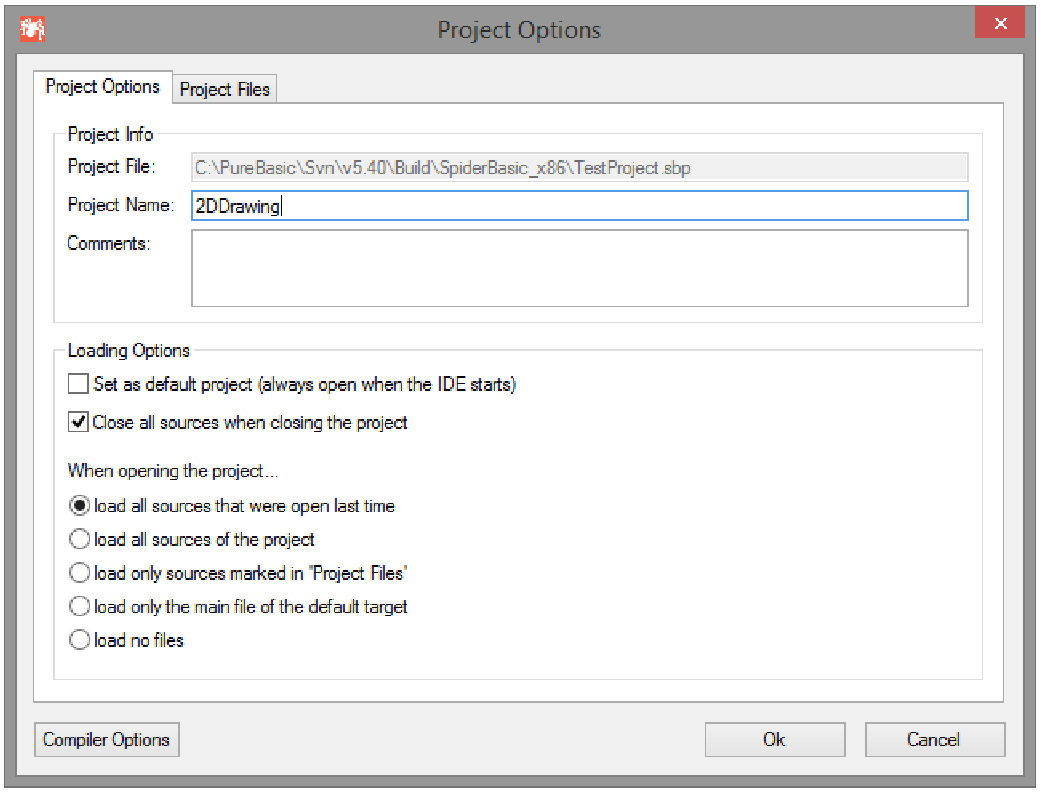

New Project... Ctrl+Shift+N
Open Project... Ctrl+Shift+O
Recent Projects
Close Project Ctrl+Shift+W
Project Options...
Add File to Project Ctrl+Shift+A
Remove File from Project Ctrl+Shift+R
Open Project Folder

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.



Project Options

Project Options | Project Files

Project Info
Project File: C:\PureBasic\Svn\v5.40\Build\SpiderBasic_x86\TestProject.sbp
Project Name: 2DDrawing
Comments:

Loading Options
☐ Set as default project (always open when the IDE starts)
☑ Close all sources when closing the project

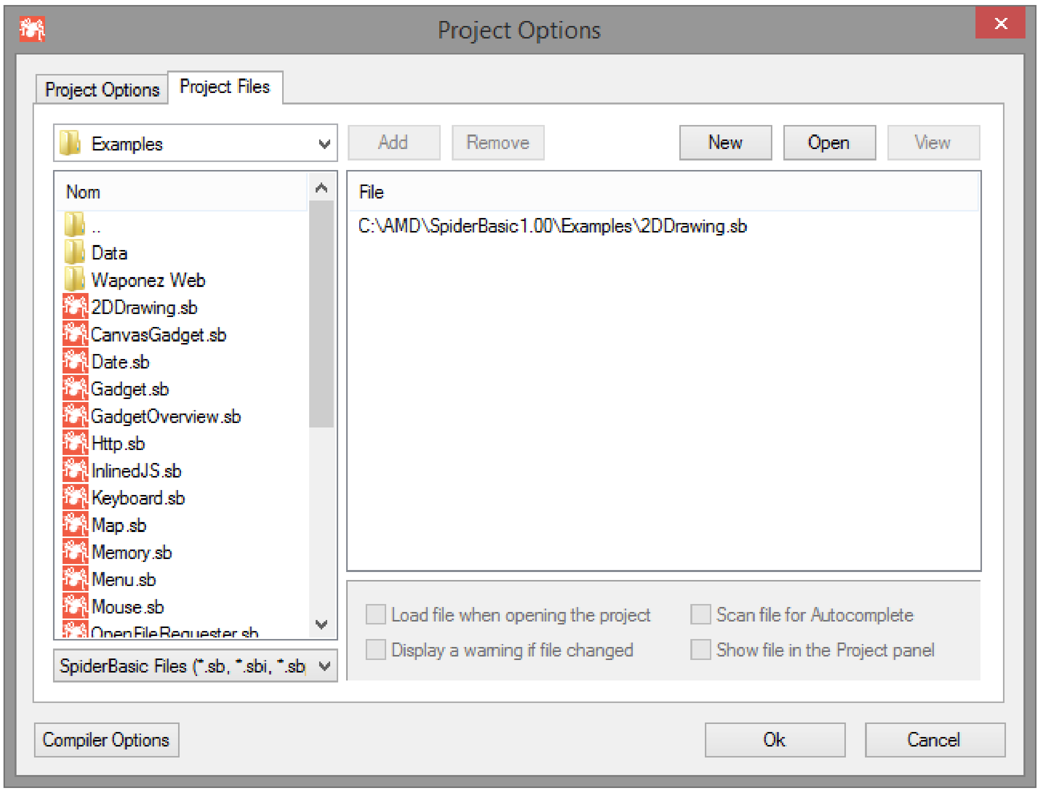
When opening the project...
● load all sources that were open last time
○ load all sources of the project
○ load only sources marked in 'Project Files'
○ load only the main file of the default target
○ load no files

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.

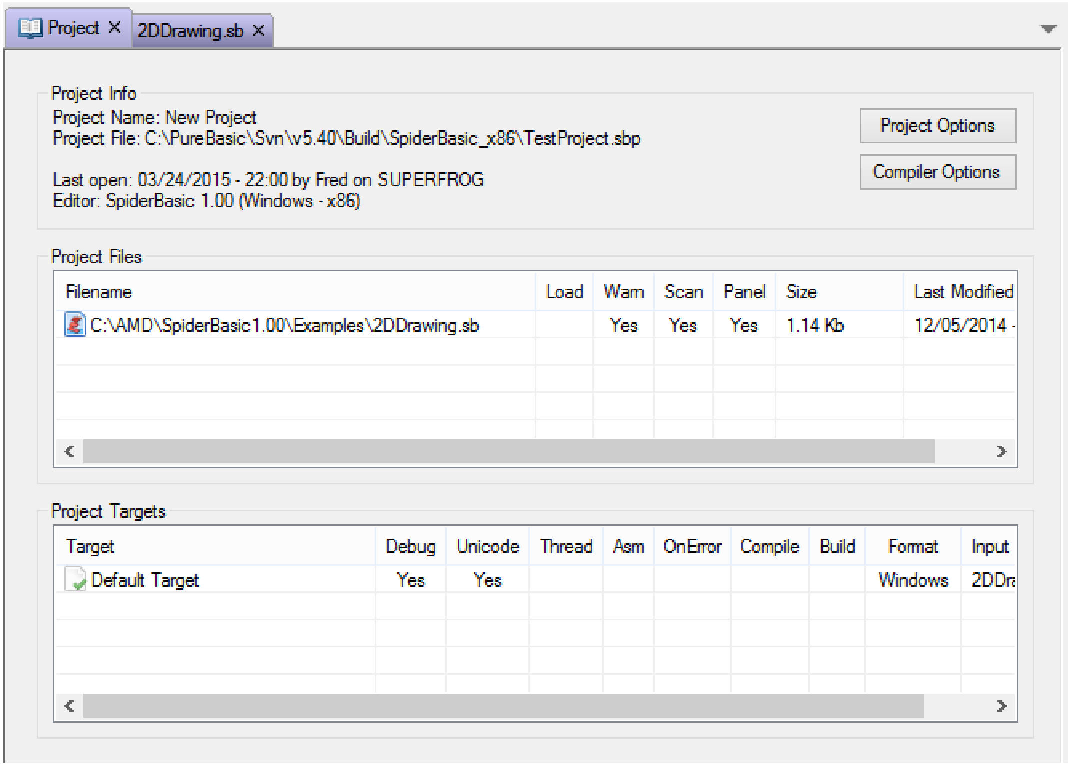
View
Opens the selected file(s) in the file list in the IDE or if they are binary files in the FileViewer.

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
Project Name: New Project
Project File: C:\PureBasic\Svn\v5.40\Build\SpiderBasic_x86\TestProject.sbp

Last open: 03/24/2015 - 22:00 by Fred on SUPERFROG
Editor: SpiderBasic 1.00 (Windows - x86)

Project Files

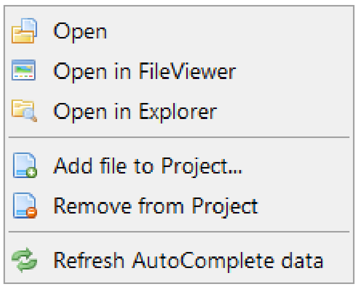
Filename	Load	Warn	Scan	Panel	Size	Last Modified
C:\AMD\SpiderBasic1.00\Examples\2DDrawing.sb		Yes	Yes	Yes	1.14 Kb	12/05/2014 -

Project Targets

Target	Debug	Unicode	Thread	Asm	OnError	Compile	Build	Format	Input
Default Target	Yes	Yes						Windows	2DDr

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

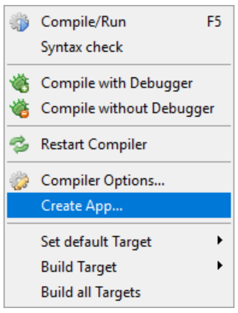
There is a sidepanel tool which allows quick access to the files belonging to the project. For more information see the built-in tools section.

Chapter 12

Compiling your programs

Compiling is easy. Just select "Compile/Run" (F5 by default) and your program will be compiled and launched in the default web browser.
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 SpiderBasic 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.

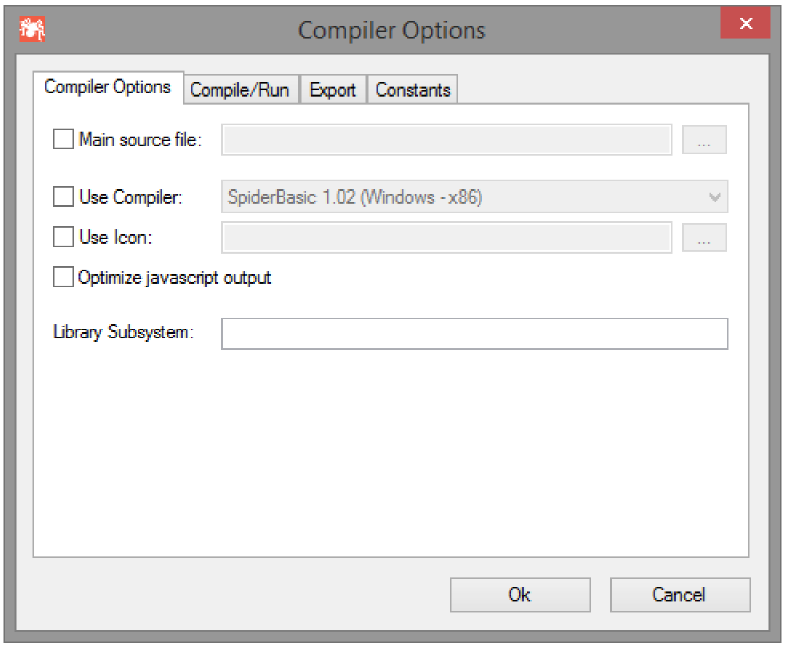
The compiler menu



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 SpiderBasic 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.
Export
Launch the export process. The export settings can be changed in the ”Compiler Options/Export” panel.
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

SpiderBasic version. This makes it easy to compile different versions of the same program (x86 and x64 or PowerPC) 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 on Windows and Linux. The same applies to the x86 and PowerPC compilers for Mac OSX. This is especially useful as this way the fast x86 IDE and debugger can be used on a Mac with an Intel processor while still compiling programs for the PowerPC processor through the slower emulation. 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
When enabled, allow to set the "favicon" file for the web application. The icon has to be in the PNG image format. This icon is usually displayed in the browser tab, near the page title.

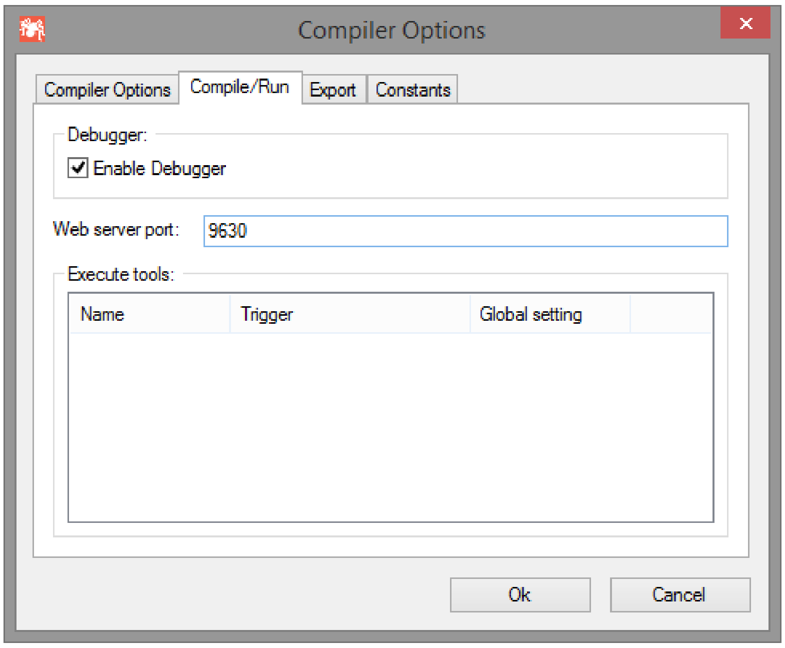
Enable DPI Aware application
This option enable DPI awareness when creating an application. That means than the canvas and images created in SpiderBasic will scale automatically if the DPI of the screen is above 100%. The functions can be used to detect which scale is currently applied when this option is enabled.

Optimize JavaScript output
When enabled, uses the Google JavaScript closure compiler to optimize the generated JavaScript code to reduce its size. A recent Java JRE needs to be installed to have this option working. The most recent JRE version can be found here: https://java.com/download.

Library Subsystem
Here you can select different subsystems for compilation. More than one subsystem can be specified, separated with space character. For more information, see subsystems .

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.


Compiler Options

Compiler Options | Compile/Run | Export | Constants

Debugger:
[✓] Enable Debugger

Web server port: 9630

Execute tools:

Name	Trigger	Global setting

Ok | Cancel

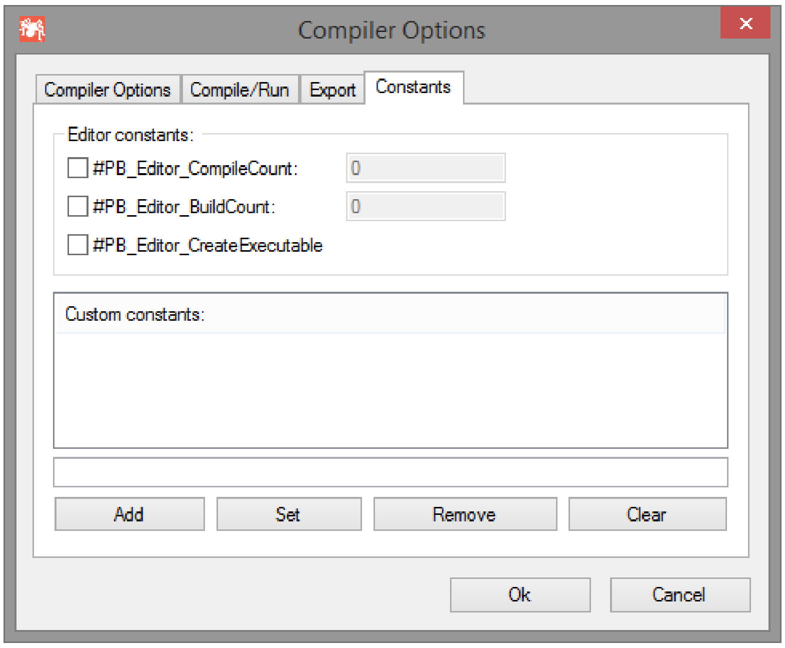
Enable Debugger
This sets the debugger state (on/off) for this source code, or if the main file option is used, for that file too. This can also be set from the debugger menu.

Web server address
This allows to set a specific web server address for this file or project. The value has to be specified as 'address:port' (example: 'localhost:8080' or 'mydomain:80'). If set to an empty value, localhost will be

used with a random dynamic port, starting from the value set in Preferences/Compiler.
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.


Compiler Options

Compiler Options | Compile/Run | Export | Constants

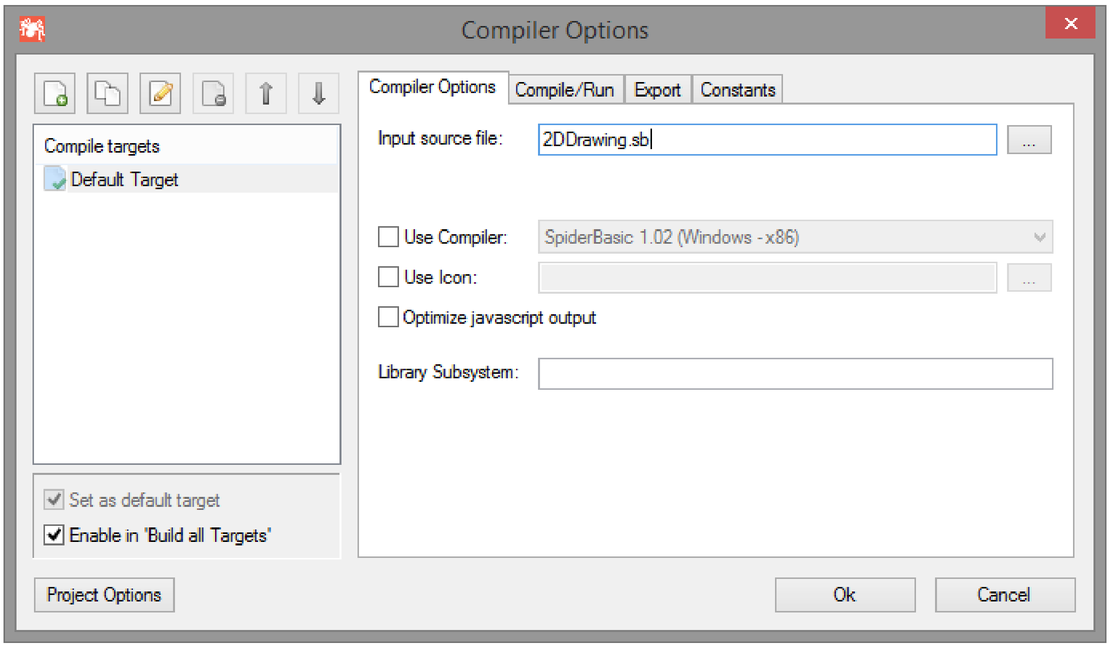
Editor constants:
[ ] #PB_Editor_CompileCount: 0
[ ] #PB_Editor_BuildCount: 0
[ ] #PB_Editor_CreateExecutable

Custom constants:

#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 constants 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.

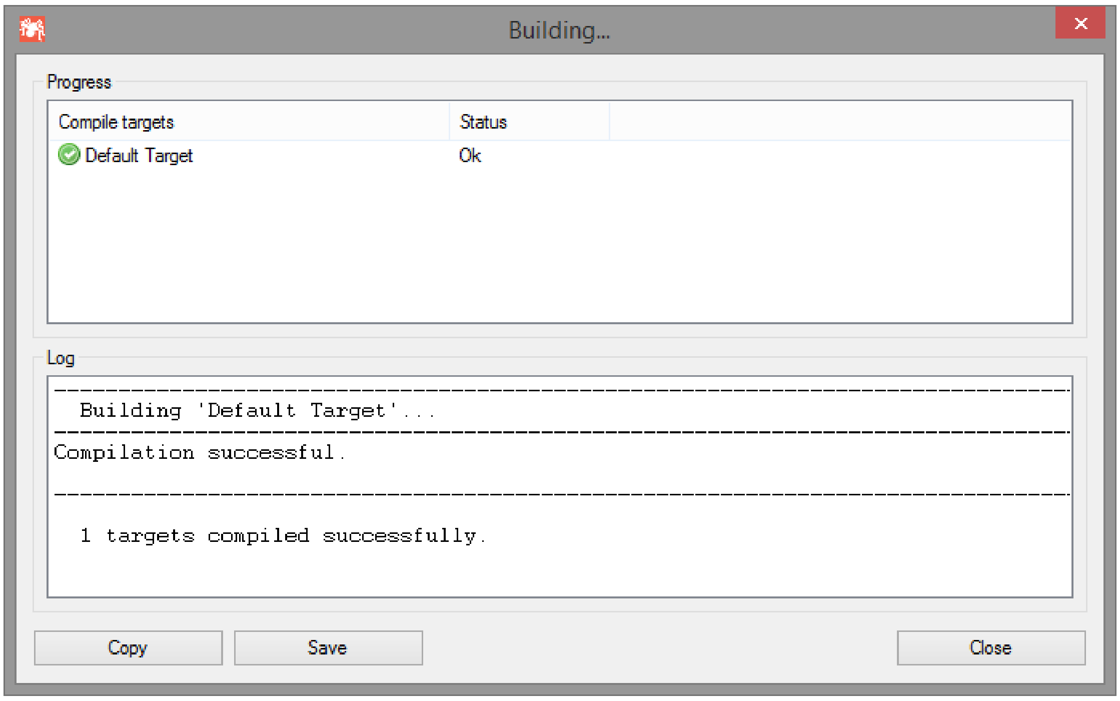
Compiler options for projects



Input source file: 2DDrawing.sb
☐ Use Compiler: SpiderBasic 1.02 (Windows - x86)
☐ Use Icon:
☐ Optimize javascript output
Library Subsystem:
☑ Set as default target
☑ Enable in 'Build all Targets'

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 is the "Input source file" on the first tab. This fields has 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



Compile targets	Status
✓ Default Target	Ok

```
--------------------------------------------------------------------------------
  Building 'Default Target'...
--------------------------------------------------------------------------------
Compilation successful.

--------------------------------------------------------------------------------

  1 targets compiled successfully.
```

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.

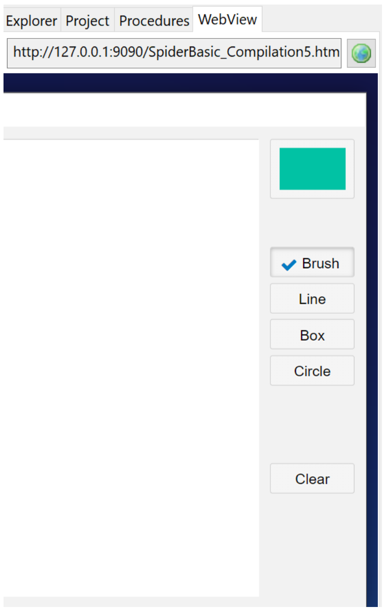
Chapter 13

Using the built-in Tools

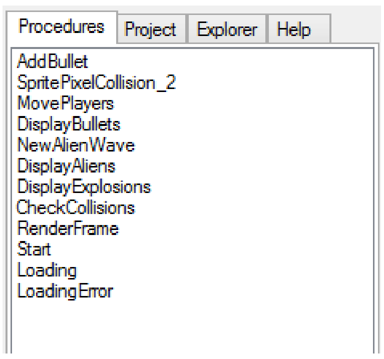
The SpiderBasic IDE comes with many building tools, to make programming tasks easier and increase your productivity. Many of them can be configured to be either accessible from the Menu as separate windows, or to be permanently displayed in the Panel on the side of the editing area.
For information on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

Tools for the Side Panel Area

WebView



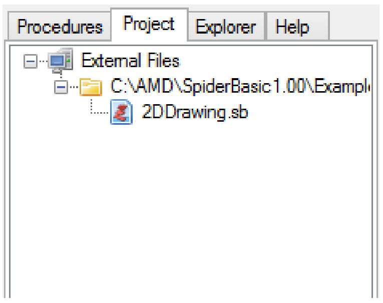
This tool displays and internal web browser to launch the SpiderBasic programs directly in the IDE. When the web view is enabled, the debugger automatically connect to it and the errors are directly reported in the IDE, on the correct file and line.
The 'earth' button is available to open the program into the default web browser.
The 'Kill Program' button (red cross in the toolbar) can be used to reset the web view content and stop the SpiderBasic program. **Procedure Browser**


Procedures
Project
Explorer
Help

AddBullet
SpritePixelCollision_2
MovePlayers
DisplayBullets
NewAlienWave
DisplayAliens
DisplayExplosions
CheckCollisions
RenderFrame
Start
Loading
LoadingError

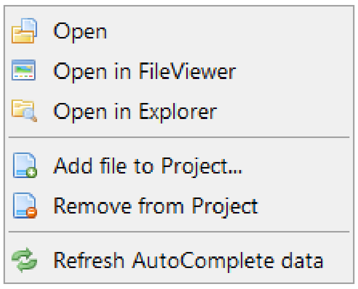
This tool displays a list of all procedures and macros declared in the current source code. By double-clicking on an entry in that list, the cursor automatically jumps to that procedure.
Macros will be marked in the list by a "+" sign before the name.
You can also place special comment marks in your 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


Procedures
Project
Explorer
Help

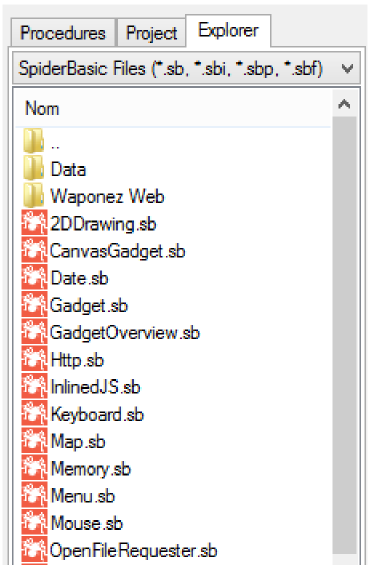
External Files
C:\AMD\SpiderBasic1.00\Exampl
2DDrawing.sb

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:



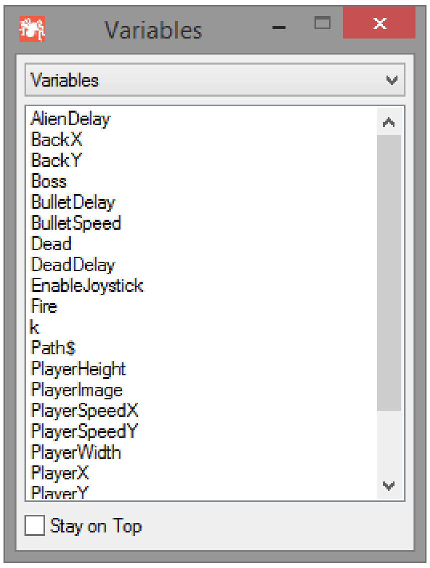
Open - Open the file in the IDE.
Open in FileViewer - Open the file in the FileViewer of the IDE.
Open in Explorer - Open the file in the operating systems file manager.
Add File to Project - Add a new file to the project.
Remove File from Project - Remove the selected file(s) from the project.
Refresh 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. SpiderBasic files (*.sb, *.sbi, *.sbp, *.sbf) will be loaded into the edit area and all other recognized files (text & binary) files will be displayed into the internal File Viewer.

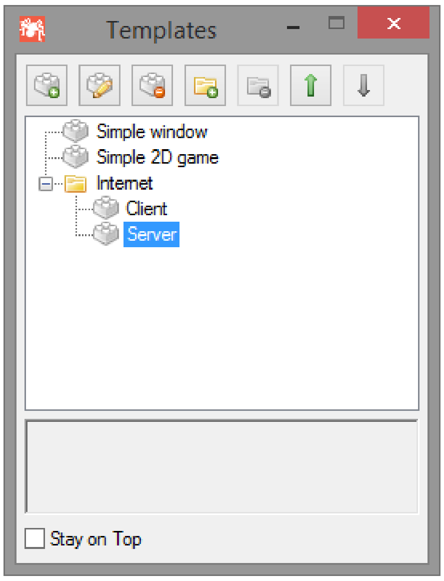
Variable Viewer



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 .

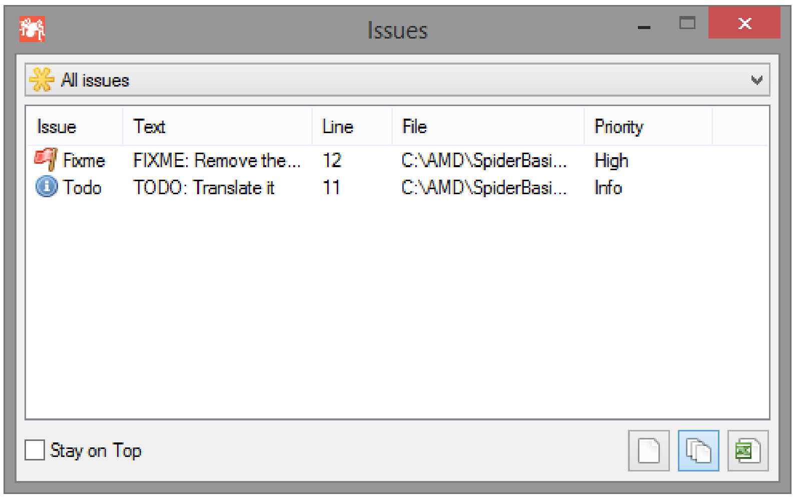
Note: The displaying of variables is somewhat limited for now. It can only detect variables explicitly declared with Define , Global , Shared , Protected or Static .

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.

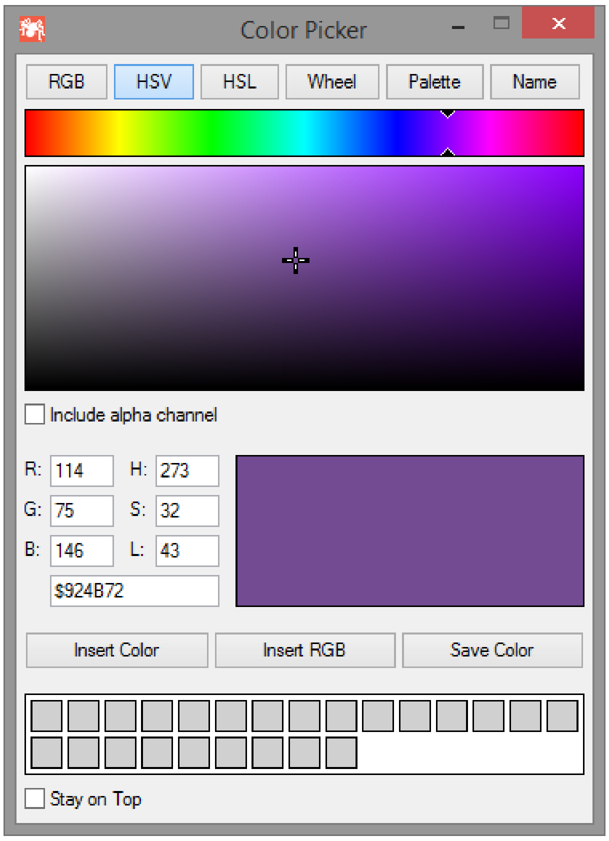
Issue Browser



Issue	Text	Line	File	Priority
Fixme	FIXME: Remove the...	12	C:\AMD\SpiderBasi...	High
Todo	TODO: Translate it	11	C:\AMD\SpiderBasi...	Info

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

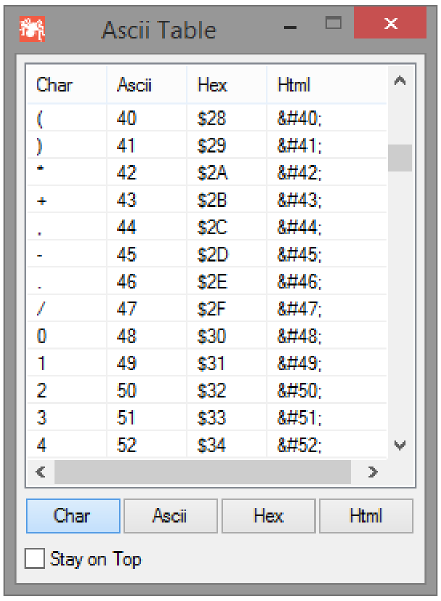


Include alpha channel: ☐

R: 114 H: 273
G: 75 S: 32
B: 146 L: 43
$924B72

Stay on Top: ☐

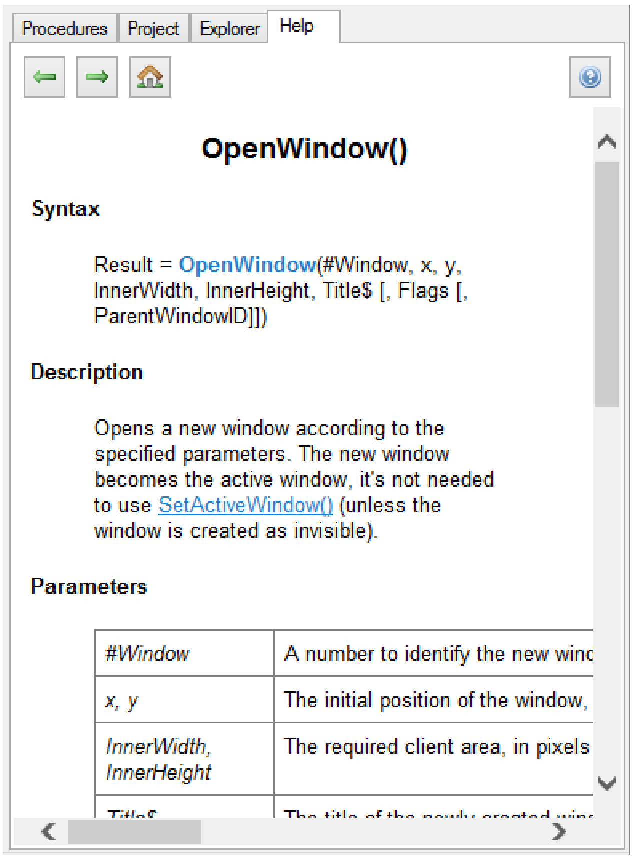
The color picker helps you to find the perfect color value for whatever task you need. The following methods of picking a color are available:
RGB: Select a color by choosing red, green and blue intensities.
HSV: Select a color by choosing hue, saturation and value.
HSL: Select a color by choosing hue, 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.
Ascii Table



Char	Ascii	Hex	Html
(	40	$28	&#40;
)	41	$29	&#41;
*	42	$2A	&#42;
+	43	$2B	&#43;
,	44	$2C	&#44;
-	45	$2D	&#45;
.	46	$2E	&#46;
/	47	$2F	&#47;
0	48	$30	&#48;
1	49	$31	&#49;
2	50	$32	&#50;
3	51	$33	&#51;
4	52	$34	&#52;

The Ascii table tool displays a table showing all the Ascii 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



OpenWindow()

Syntax

Result = OpenWindow(#Window, x, y, InnerWidth, InnerHeight, Title$ [, Flags [, ParentWindowID]])

Description

Opens a new window according to the specified parameters. The new window becomes the active window, it's not needed to use SetActiveWindow() (unless the window is created as invisible).

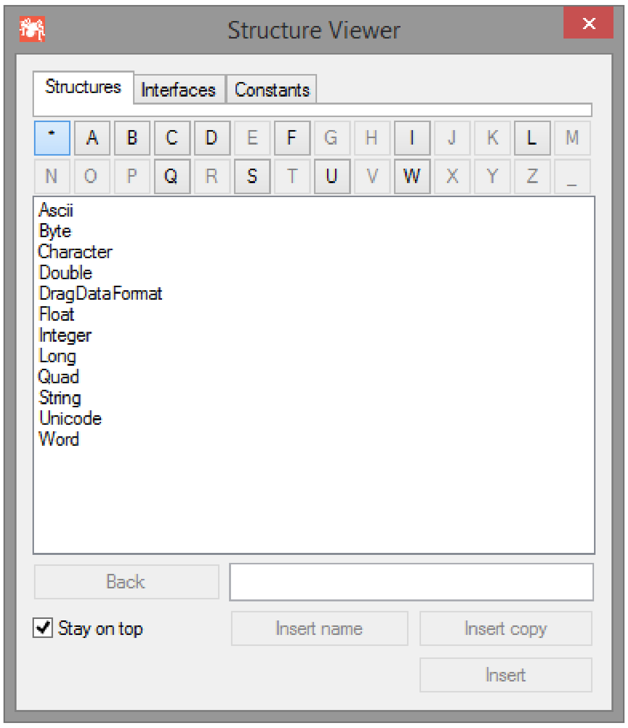
Parameters

#Window	A number to identify the new winc
x, y	The initial position of the window,
InnerWidth, InnerHeight	The required client area, in pixels

The Help Tool is an alternative viewer for the reference guide . It can be used to view the SpiderBasic 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

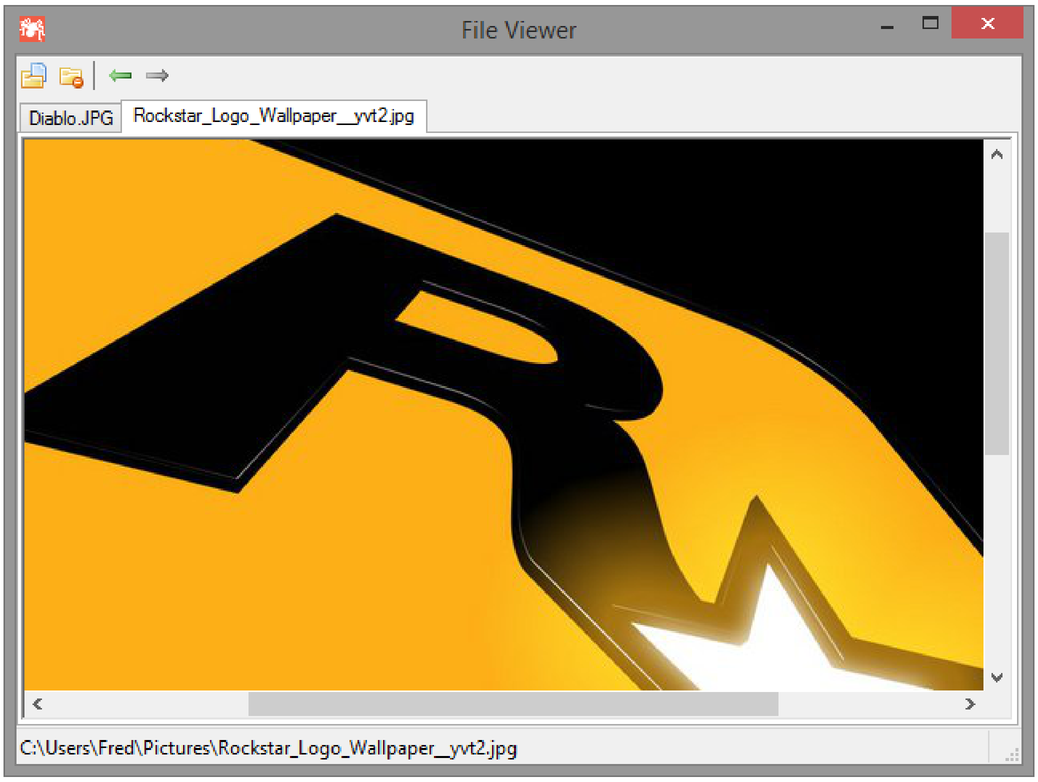
Structure Viewer



Ascii
Byte
Character
Double
DragDataFormat
Float
Integer
Long
Quad
String
Unicode
Word

The structure viewer allows you to view all the Structures, Interfaces and Constants predefined in SpiderBasic. 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.

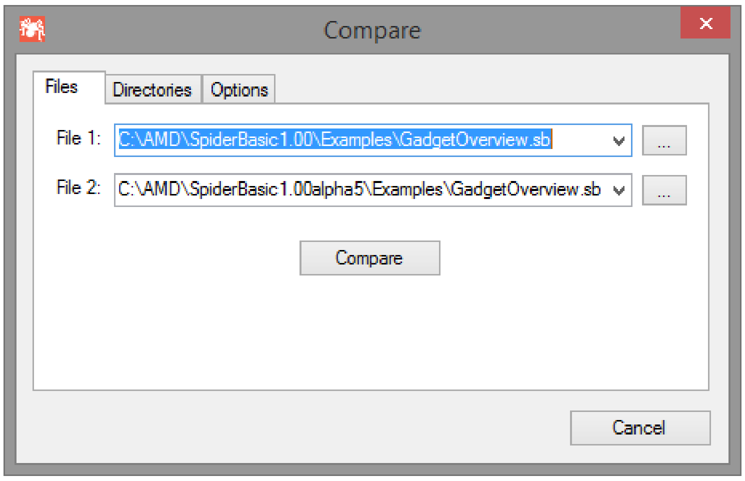
File Viewer



C:\Users\Fred\Pictures\Rockstar_Logo_Wallpaper__yvt2.jpg

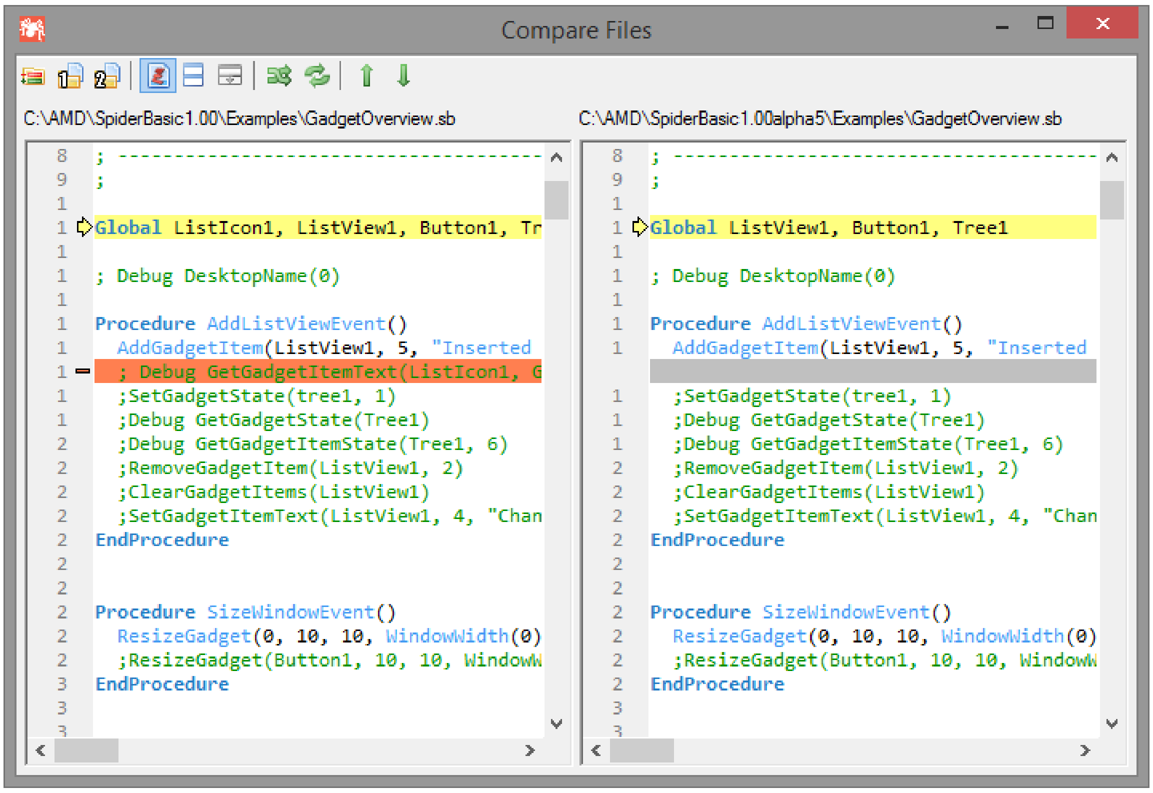
The internal file viewer allows you do display certain types of files. Text files, images and web pages (windows only). Any unknown file type will be displayed in a hex-viewer. The "Open" button opens a new file, the "X button" closes it and the arrows can be used to navigate through the open files.
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



File 1: C:\AMD\SpiderBasic1.00\Examples\GadgetOverview.sb
File 2: C:\AMD\SpiderBasic1.00alpha5\Examples\GadgetOverview.sb

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.



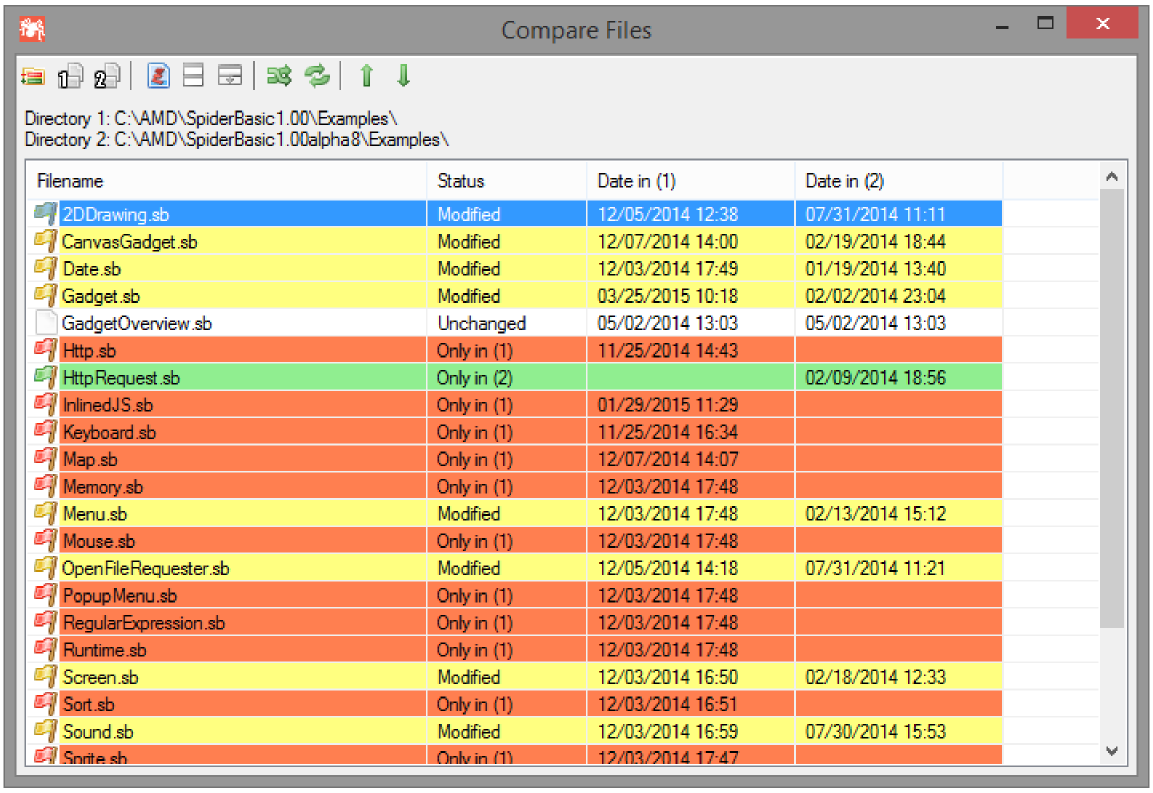
C:\AMD\SpiderBasic1.00\Examples\GadgetOverview.sb

```
; ------------------------------------
;

Global ListIcon1, ListView1, Button1, Tr

; Debug DesktopName(0)

Procedure AddListViewEvent()
  AddGadgetItem(ListView1, 5, "Inserted
  ; Debug GetGadgetItemText(ListIcon1, G
  ;SetGadgetState(tree1, 1)
  ;Debug GetGadgetState(Tree1)
  ;Debug GetGadgetItemState(Tree1, 6)
  ;RemoveGadgetItem(ListView1, 2)
  ;ClearGadgetItems(ListView1)
  ;SetGadgetItemText(ListView1, 4, "Chan
EndProcedure


Procedure SizeWindowEvent()
  ResizeGadget(0, 10, 10, WindowWidth(0)
  ;ResizeGadget(Button1, 10, 10, WindowW
EndProcedure
```

C:\AMD\SpiderBasic1.00alpha5\Examples\GadgetOverview.sb

```
; ------------------------------------
;

Global ListView1, Button1, Tree1

; Debug DesktopName(0)

Procedure AddListViewEvent()
  AddGadgetItem(ListView1, 5, "Inserted

  ;SetGadgetState(tree1, 1)
  ;Debug GetGadgetState(Tree1)
  ;Debug GetGadgetItemState(Tree1, 6)
  ;RemoveGadgetItem(ListView1, 2)
  ;ClearGadgetItems(ListView1)
  ;SetGadgetItemText(ListView1, 4, "Chan
EndProcedure


Procedure SizeWindowEvent()
  ResizeGadget(0, 10, 10, WindowWidth(0)
  ;ResizeGadget(Button1, 10, 10, WindowW
EndProcedure
```

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.



Directory 1: C:\AMD\SpiderBasic1.00\Examples\
Directory 2: C:\AMD\SpiderBasic1.00alpha8\Examples\

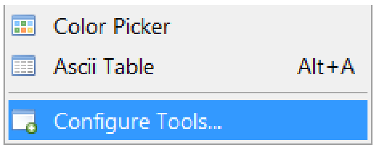
Filename	Status	Date in (1)	Date in (2)
2DDrawing.sb	Modified	12/05/2014 12:38	07/31/2014 11:11
CanvasGadget.sb	Modified	12/07/2014 14:00	02/19/2014 18:44
Date.sb	Modified	12/03/2014 17:49	01/19/2014 13:40
Gadget.sb	Modified	03/25/2015 10:18	02/02/2014 23:04
GadgetOverview.sb	Unchanged	05/02/2014 13:03	05/02/2014 13:03
Http.sb	Only in (1)	11/25/2014 14:43	
HttpRequest.sb	Only in (2)		02/09/2014 18:56
InlinedJS.sb	Only in (1)	01/29/2015 11:29	
Keyboard.sb	Only in (1)	11/25/2014 16:34	
Map.sb	Only in (1)	12/07/2014 14:07	
Memory.sb	Only in (1)	12/03/2014 17:48	
Menu.sb	Modified	12/03/2014 17:48	02/13/2014 15:12
Mouse.sb	Only in (1)	12/03/2014 17:48	
OpenFileRequester.sb	Modified	12/05/2014 14:18	07/31/2014 11:21
PopupMenu.sb	Only in (1)	12/03/2014 17:48	
RegularExpression.sb	Only in (1)	12/03/2014 17:48	
Runtime.sb	Only in (1)	12/03/2014 17:48	
Screen.sb	Modified	12/03/2014 16:50	02/18/2014 12:33
Sort.sb	Only in (1)	12/03/2014 16:51	
Sound.sb	Modified	12/03/2014 16:59	07/30/2014 15:53
Sprite.sb	Only in (1)	12/03/2014 17:47	

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.

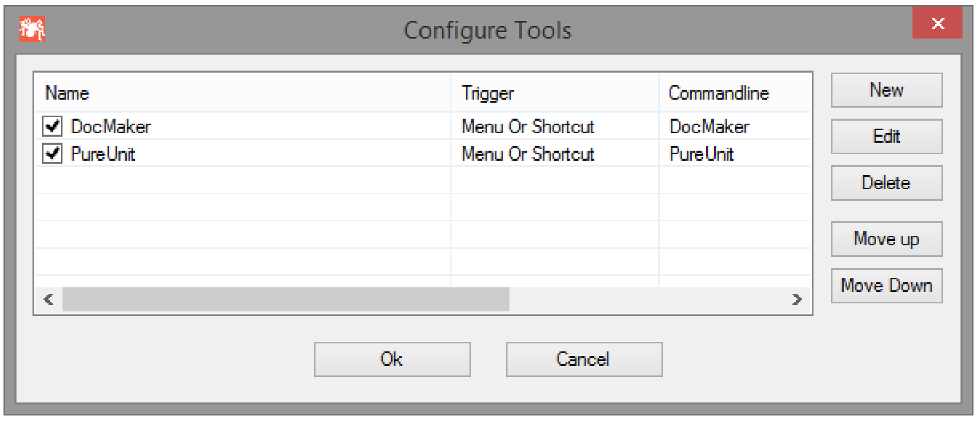
Chapter 14

Using external tools

The SpiderBasic 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 SpiderBasic 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.

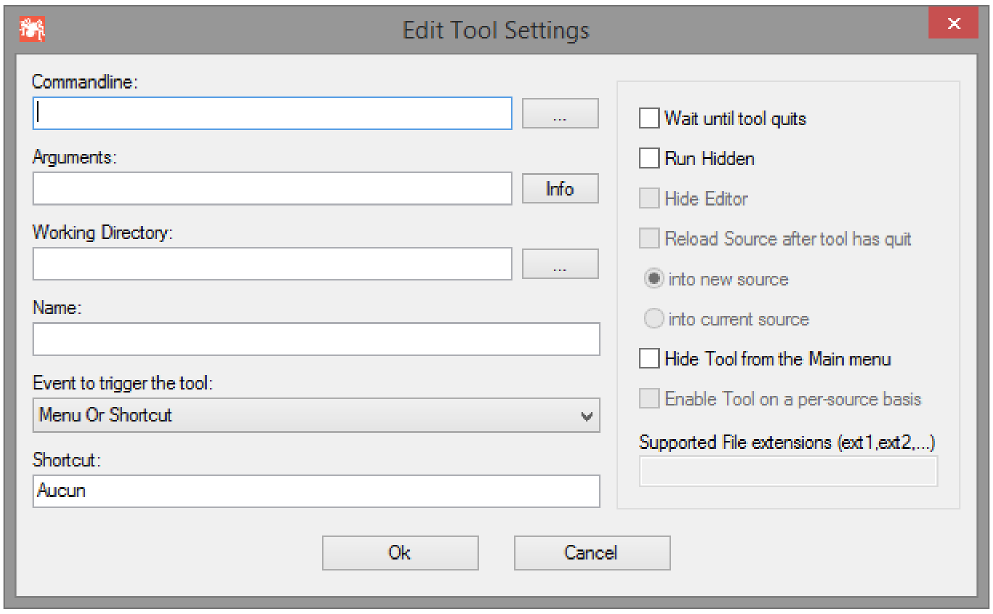


Name	Trigger	Commandline
✓ DocMaker	Menu Or Shortcut	DocMaker
✓ PureUnit	Menu Or Shortcut	PureUnit

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

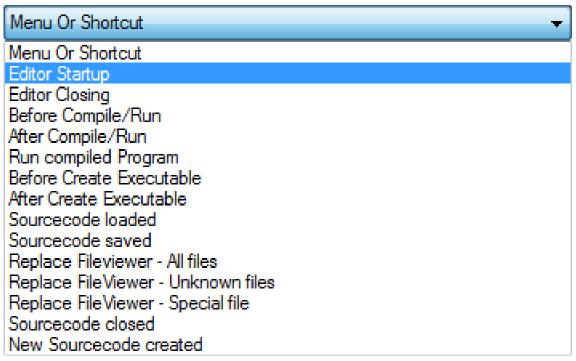
The basic things you need to set is the command-line of the program to run, and a name for it in the Tools list/Menu. Everything else is optional.



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.
%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 spiderbasic 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
Menu Or Shortcut
Editor Startup
Editor Closing
Before Compile/Run
After Compile/Run
Run compiled Program
Before Create Executable
After Create Executable
Sourcecode loaded
Sourcecode saved
Replace Fileviewer - All files
Replace FileViewer - Unknown files
Replace FileViewer - Special file
Sourcecode closed
New Sourcecode created

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.

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.
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
```

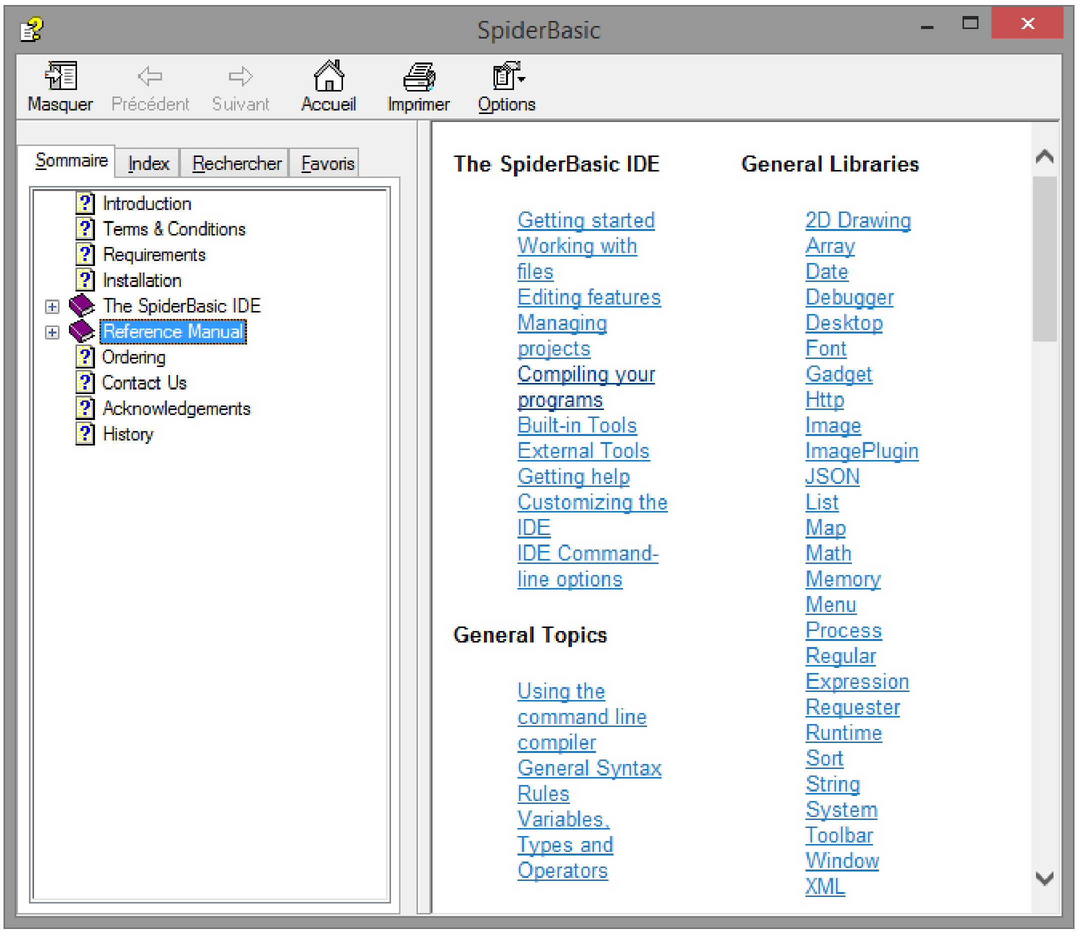
When the %TEMPFILE or %COMPILEFILE tokens are used, the IDE appends the compiler options as a comment to the end of the created temporary file, even if the user did choose to not save the options there when saving a source code.
This enables your tool to read the compiler settings for this file, and take them into account for the actions your carries out.

Chapter 15

Getting Help

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

Quick access to the reference guide



The SpiderBasic IDE

Getting started
Working with files
Editing features
Managing projects
Compiling your programs
Built-in Tools
External Tools
Getting help
Customizing the IDE
IDE Command-line options

General Topics

Using the command line compiler
General Syntax Rules
Variables, Types and Operators

General Libraries

2D Drawing
Array
Date
Debugger
Desktop
Font
Gadget
Http
Image
ImagePlugin
JSON
List
Map
Math
Memory
Menu
Process
Regular Expression
Requester
Runtime
Sort
String
System
Toolbar
Window
XML

By pressing the help shortcut (F1 by default) or selecting the ”Help...” command from the Help menu while the mouse cursor is over a SpiderBasic 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 .

Accessing external helpfiles from the IDE

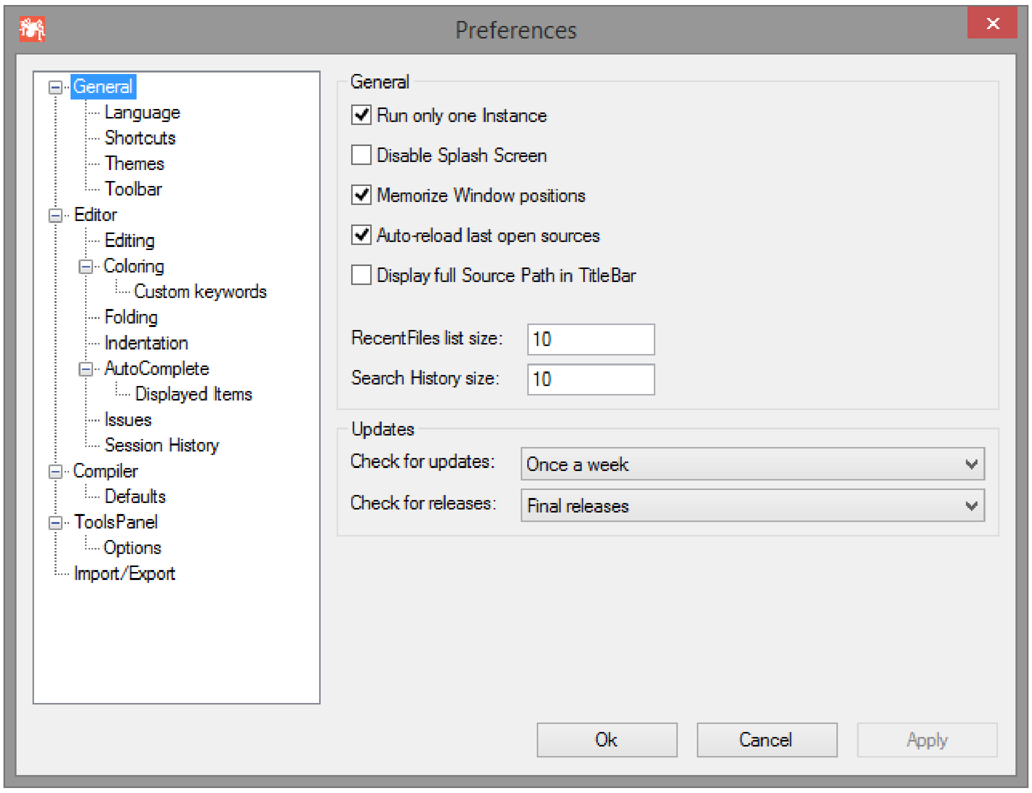
If you have other helpfiles you wish to be able to access from the IDE, then create a ”Help” subdirectory in your SpiderBasic folder and copy them to it. These files will appear in the ”External Help” submenu of the Help menu, and in the popupmenu you get when right-clicking in the editing area. Chm 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 SpiderBasic. Now if you click the file in the ”external help” menu, it will be opened in that external tool.

Chapter 16

Customizing the IDE

The SpiderBasic 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



General

- [✓] Run only one Instance
- [ ] Disable Splash Screen
- [✓] Memorize Window positions
- [✓] Auto-reload last open sources
- [ ] Display full Source Path in TitleBar

RecentFiles list size: 10
Search History size: 10

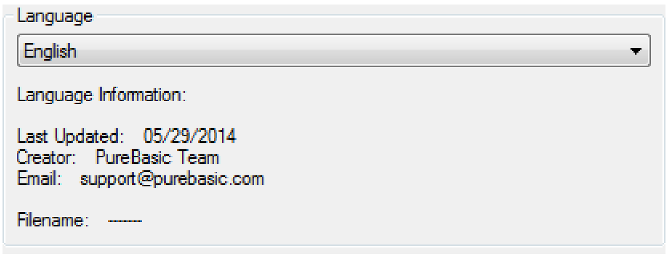
Updates

Check for updates: Once a week
Check for releases: Final releases

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.
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 spiderbasic.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


Language
English

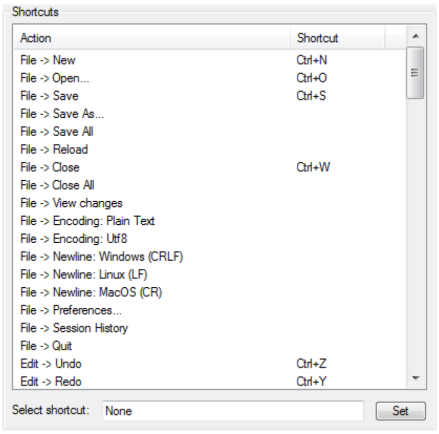
Language Information:

Last Updated: 05/29/2014
Creator: PureBasic Team
Email: support@purebasic.com

Filename: -------

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


Shortcuts

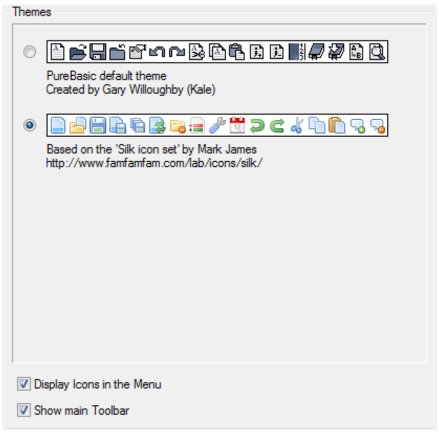
Action	Shortcut
File -> New	Ctrl+N
File -> Open...	Ctrl+O
File -> Save	Ctrl+S
File -> Save As...	
File -> Save All	
File -> Reload	
File -> Close	Ctrl+W
File -> Close All	
File -> View changes	
File -> Encoding: Plain Text	
File -> Encoding: Utf8	
File -> Newline: Windows (CRLF)	
File -> Newline: Linux (LF)	
File -> Newline: MacOS (CR)	
File -> Preferences...	
File -> Session History	
File -> Quit	
Edit -> Undo	Ctrl+Z
Edit -> Redo	Ctrl+Y

Select shortcut: None

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


Themes

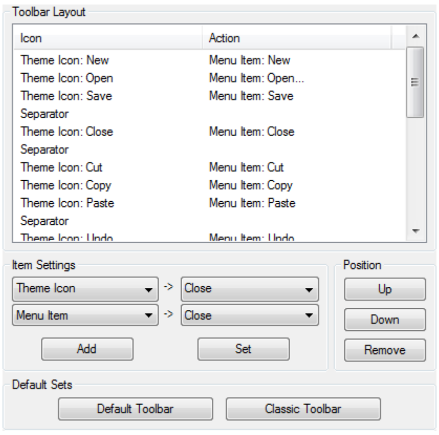
( ) PureBasic default theme
Created by Gary Willoughby (Kale)

(●) Based on the 'Silk icon set' by Mark James
http://www.famfamfam.com/lab/icons/silk/

[✓] Display Icons in the Menu
[✓] Show main Toolbar

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 SpiderBasic 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


Toolbar Layout

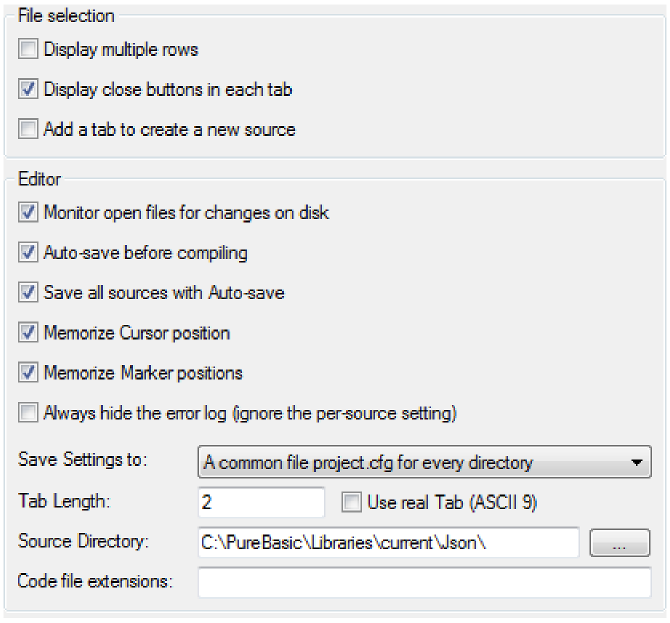
Icon	Action
Theme Icon: New	Menu Item: New
Theme Icon: Open	Menu Item: Open...
Theme Icon: Save	Menu Item: Save
Separator	
Theme Icon: Close	Menu Item: Close
Separator	
Theme Icon: Cut	Menu Item: Cut
Theme Icon: Copy	Menu Item: Copy
Theme Icon: Paste	Menu Item: Paste
Separator	
Theme Icon: Undo	Menu Item: Undo

Item Settings
Theme Icon -> Close
Menu Item -> Close

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


File selection
- [ ] Display multiple rows
- [✓] Display close buttons in each tab
- [ ] Add a tab to create a new source

Editor
- [✓] Monitor open files for changes on disk
- [✓] Auto-save before compiling
- [✓] Save all sources with Auto-save
- [✓] Memorize Cursor position
- [✓] Memorize Marker positions
- [ ] Always hide the error log (ignore the per-source setting)

Save Settings to: A common file project.cfg for every directory
Tab Length: 2 [ ] Use real Tab (ASCII 9)
Source Directory: C:\PureBasic\Libraries\current\Json\
Code file extensions:

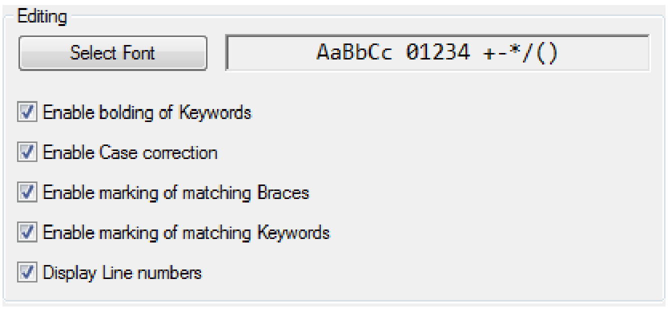
Settings that affect the management of the source codes.
Monitor open files for changes on disk
Monitors all open files for changes that are made to the files on disk while they are edited in the IDE. If modifications are made by other programs, a warning is displayed with the choice to reload the file 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>.sb.cfg
Creates a .sb.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.

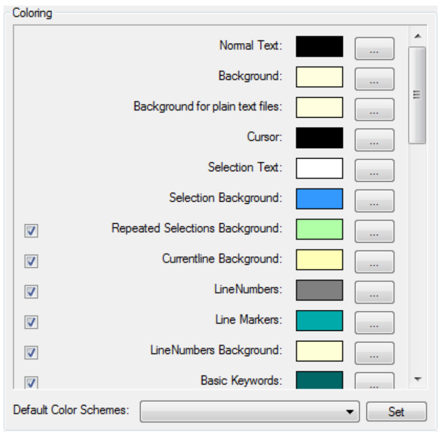
Tab Length
Allows to specify how many spaces are inserted each time you press the Tab key.
Use real Tab (Ascii 9)
If set, the tab key inserts a real tab character instead of spaces. If not set, there are spaces inserted when Tab is pressed.
Note that if real tab is used, the ”Tab Length” option specifies 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 (sb, sbi or sbf 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. ”sbx, xyz”) of extensions to recognize.

Editor - Editing


Editing
Select Font
AaBbCc 01234 +-*/()
✓ Enable bolding of Keywords
✓ Enable Case correction
✓ Enable marking of matching Braces
✓ Enable marking of matching Keywords
✓ Display Line numbers

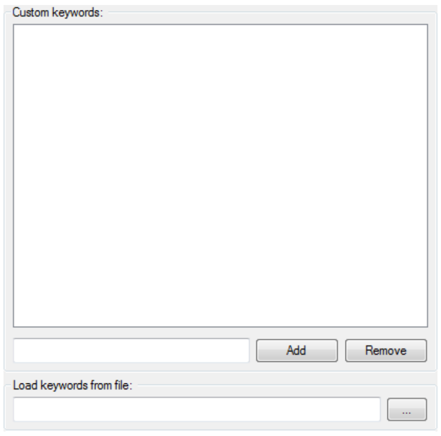
Use ”Select Font” to change the font used to display the source code. To ensure a good view of the source code, it should be a fixed-size font, and possibly even one where bold characters have the same size as non-bold ones.
Enable bolding of keywords
If your font does not display bold characters in 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 SpiderBasic keywords, SpiderBasic 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



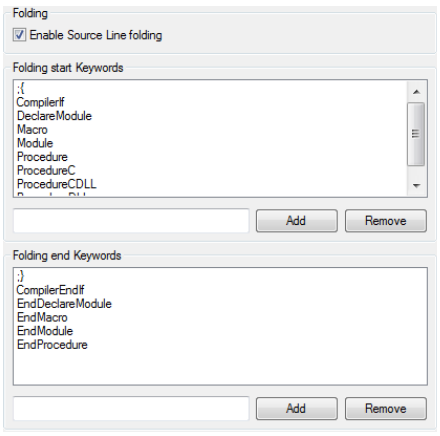
Here you can change the color settings for the syntax coloring, as well as the debugger marks. Default color schemes can be selected from the box on the bottom, and also modified after they have been set. Individual color settings can be disabled by use of the checkboxes.
Note: The 'Accessibility' color 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.

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 SpiderBasic 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


Folding

[✓] Enable Source Line folding

Folding start Keywords

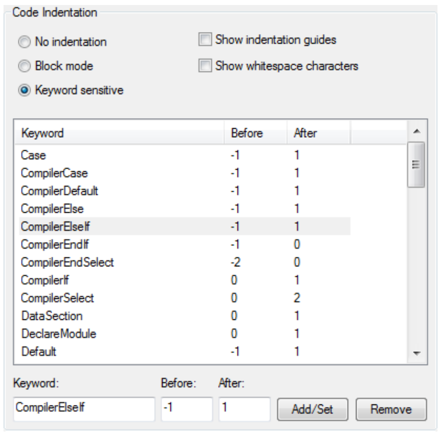
;{
CompilerIf
DeclareModule
Macro
Module
Procedure
ProcedureC
ProcedureCDLL

Folding end Keywords

;}
CompilerEndIf
EndDeclareModule
EndMacro
EndModule
EndProcedure

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


Code Indentation

○ No indentation
○ Block mode
[● Keyword sensitive]

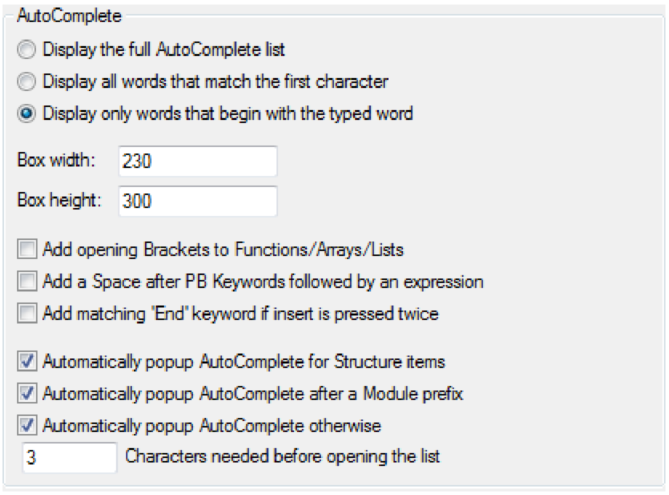
☐ Show indentation guides
☐ Show whitespace characters

Keyword	Before	After
Case	-1	1
CompilerCase	-1	1
CompilerDefault	-1	1
CompilerElse	-1	1
CompilerElseIf	-1	1
CompilerEndIf	-1	0
CompilerEndSelect	-2	0
CompilerIf	0	1
CompilerSelect	0	2
DataSection	0	1
DeclareModule	0	1
Default	-1	1

Keyword: CompilerElseIf Before: -1 After: 1

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


AutoComplete

- ( ) Display the full AutoComplete list
- ( ) Display all words that match the first character
- (●) Display only words that begin with the typed word

Box width: 230

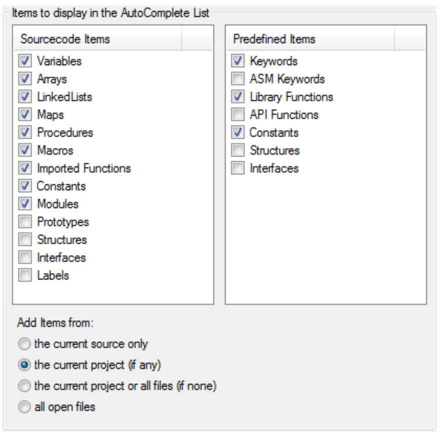
Box height: 300

- [ ] Add opening Brackets to Functions/Arrays/Lists
- [ ] Add a Space after PB Keywords followed by an expression
- [ ] Add matching 'End' keyword if insert is pressed twice
- [✓] Automatically popup AutoComplete for Structure items
- [✓] Automatically popup AutoComplete after a Module prefix
- [✓] Automatically popup AutoComplete otherwise

3 Characters needed before opening the list

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 Tab/Enter is pressed twice
If you press Tab or Enter twice, it will insert the corresponding end keyword (for example "EndSelect" to "Select" or "EndIf " to "If") to the keyword you have just inserted. 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

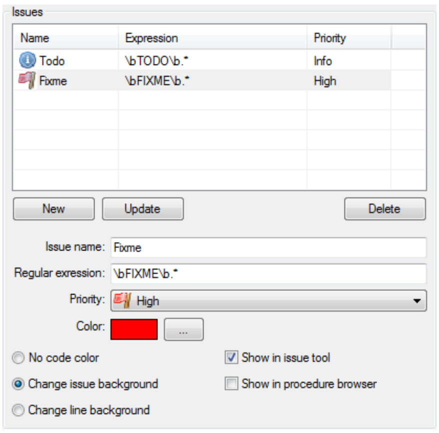

Items to display in the AutoComplete List

Sourcecode Items	Predefined Items
[✓] Variables	[✓] Keywords
[✓] Arrays	[ ] ASM Keywords
[✓] LinkedLists	[✓] Library Functions
[✓] Maps	[ ] API Functions
[✓] Procedures	[✓] Constants
[✓] Macros	[ ] Structures
[✓] Imported Functions	[ ] Interfaces
[✓] Constants	
[✓] Modules	
[ ] Prototypes	
[ ] Structures	
[ ] Interfaces	
[ ] Labels	

Add Items from:
- the current source only
- [✓ the current project (if any)]
- the current project or all files (if none)
- all open files

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 SpiderBasic, such as the SpiderBasic 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


Issues

Name	Expression	Priority
Todo	\bTODO\b.*	Info
Fixme	\bFIXME\b.*	High

Issue name: Fixme
Regular exression: \bFIXME\b.*
Priority: High
Color: 🟥

- No code color
- [✓ Change issue background]
- Change line background

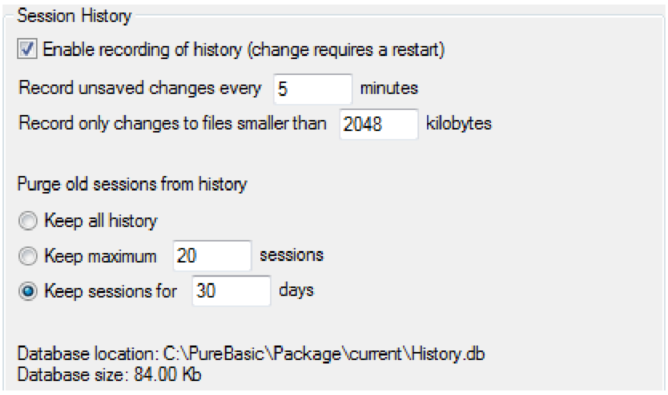
[✓] Show in issue tool
[ ] Show in procedure browser

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:

<u>Issue name</u>
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.
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


Session History

[✓] Enable recording of history (change requires a restart)

Record unsaved changes every 5 minutes

Record only changes to files smaller than 2048 kilobytes

Purge old sessions from history

○ Keep all history

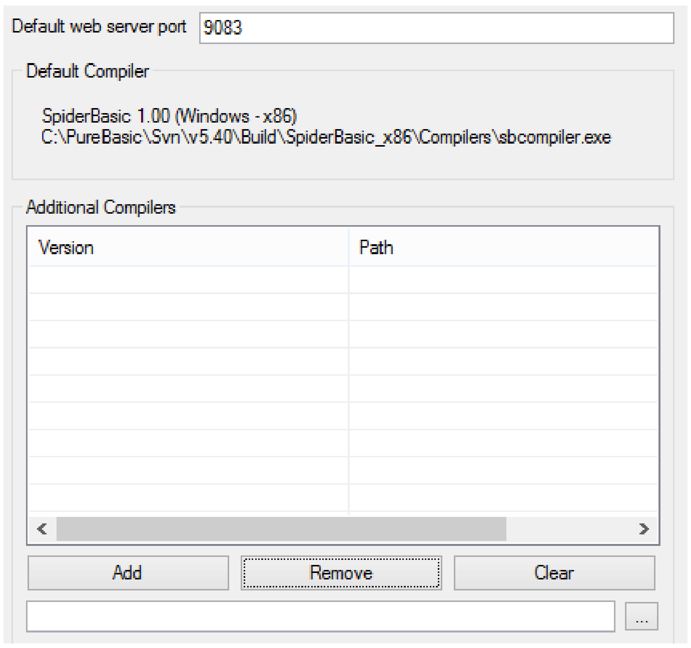
○ Keep maximum 20 sessions

● Keep sessions for 30 days

Database location: C:\PureBasic\Package\current\History.db
Database size: 84.00 Kb

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


Default web server port: 9083

Default Compiler

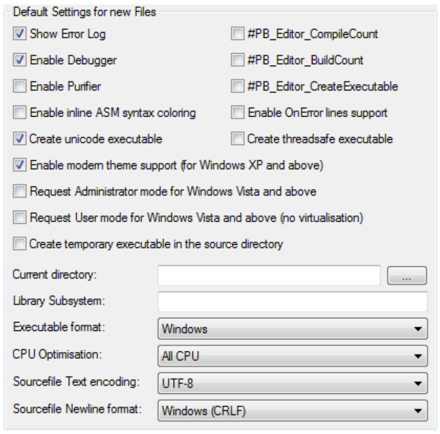
SpiderBasic 1.00 (Windows - x86)
C:\PureBasic\Svn\v5.40\Build\SpiderBasic_x86\Compilers\sbcompiler.exe

Additional Compilers

Version	Path

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 SpiderBasic compiler 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

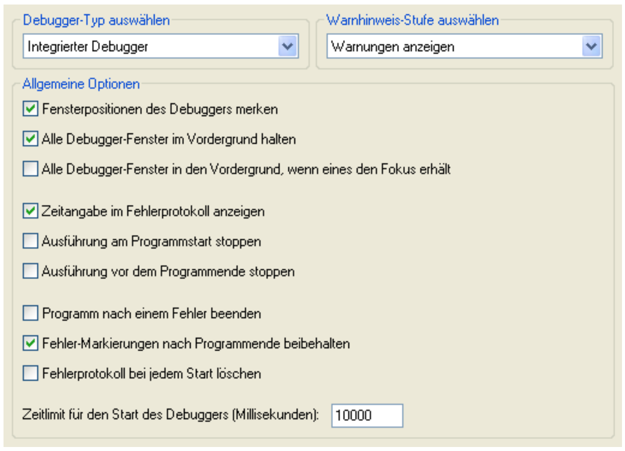

Default Settings for new Files

- [✓] Show Error Log
- [✓] Enable Debugger
- [ ] Enable Purifier
- [ ] Enable inline ASM syntax coloring
- [✓] Create unicode executable
- [ ] #PB_Editor_CompileCount
- [ ] #PB_Editor_BuildCount
- [ ] #PB_Editor_CreateExecutable
- [ ] Enable OnError lines support
- [ ] Create threadsafe executable
- [✓] Enable modern theme support (for Windows XP and above)
- [ ] Request Administrator mode for Windows Vista and above
- [ ] Request User mode for Windows Vista and above (no virtualisation)
- [ ] Create temporary executable in the source directory

Current directory:	
Library Subsystem:	
Executable format:	Windows
CPU Optimisation:	All CPU
Sourcefile Text encoding:	UTF-8
Sourcefile Newline format:	Windows (CRLF)

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



Debugger-Typ auswählen: Integrierter Debugger

Warnhinweis-Stufe auswählen: Warnungen anzeigen

Allgemeine Optionen

- ✓ Fensterpositionen des Debuggers merken
- ✓ Alle Debugger-Fenster im Vordergrund halten
- ☐ Alle Debugger-Fenster in den Vordergrund, wenn eines den Fokus erhält
- ✓ Zeitangabe im Fehlerprotokoll anzeigen
- ☐ Ausführung am Programmstart stoppen
- ☐ Ausführung vor dem Programmende stoppen
- ☐ Programm nach einem Fehler beenden
- ✓ Fehler-Markierungen nach Programmende beibehalten
- ☐ Fehlerprotokoll bei jedem Start löschen

Zeitlimit für den Start des Debuggers (Millisekunden): 10000

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.

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.

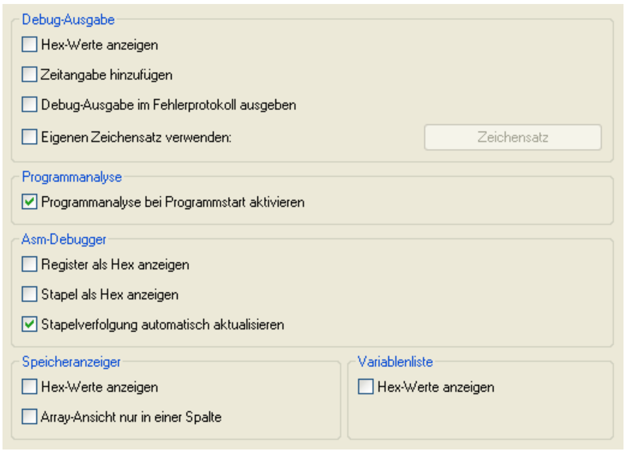
Keep Error marks after program end
Does not remove the lines marked with errors when the program ends. This gives the opportunity to still see where an error occurred while editing the code again.
The marks can be manually removed with the "Clear error marks" command in the "Error log" submenu of the 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-Ausgabe
- [ ] Hex-Werte anzeigen
- [ ] Zeitangabe hinzufügen
- [ ] Debug-Ausgabe im Fehlerprotokoll ausgeben
- [ ] Eigenen Zeichensatz verwenden:

Programmanalyse
- [✓] Programmanalyse bei Programmstart aktivieren

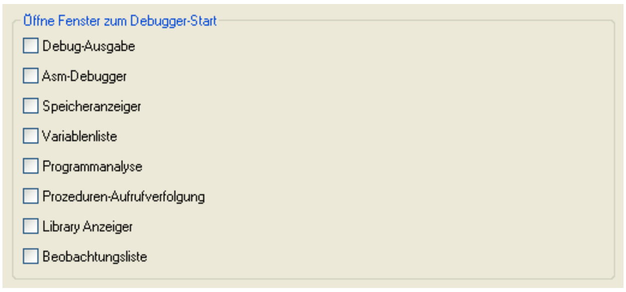
Asm-Debugger
- [ ] Register als Hex anzeigen
- [ ] Stapel als Hex anzeigen
- [✓] Stapelverfolgung automatisch aktualisieren

Speicheranzeiger
- [ ] Hex-Werte anzeigen
- [ ] Array-Ansicht nur in einer Spalte

Variablenliste
- [ ] Hex-Werte anzeigen

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.

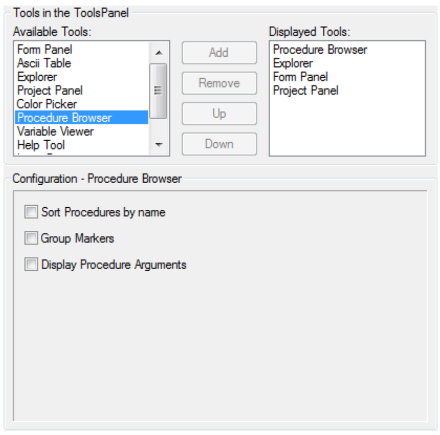
Debugger - Default Windows



Öffne Fenster zum Debugger-Start
- [ ] Debug-Ausgabe
- [ ] Asm-Debugger
- [ ] Speicheranzeiger
- [ ] Variablenliste
- [ ] Programmanalyse
- [ ] Prozeduren-Aufrufverfolgung
- [ ] Library Anzeiger
- [ ] Beobachtungsliste

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

Tools Panel


Tools in the ToolsPanel

Available Tools:
- Form Panel
- Ascii Table
- Explorer
- Project Panel
- Color Picker
- Procedure Browser
- Variable Viewer
- Help Tool

Add | Remove | Up | Down

Displayed Tools:
- Procedure Browser
- Explorer
- Form Panel
- Project Panel

Configuration - Procedure Browser

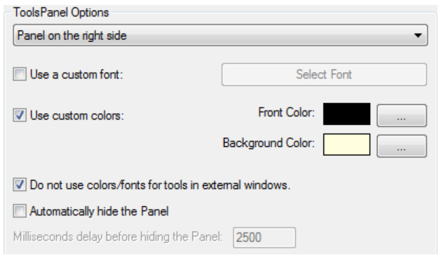
☐ Sort Procedures by name

☐ Group Markers

☐ Display Procedure Arguments

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


ToolsPanel Options

Panel on the right side

☐ Use a custom font: — Select Font

☑ Use custom colors: — Front Color: ■ ... — Background Color: □ ...

☑ Do not use colors/fonts for tools in external windows.

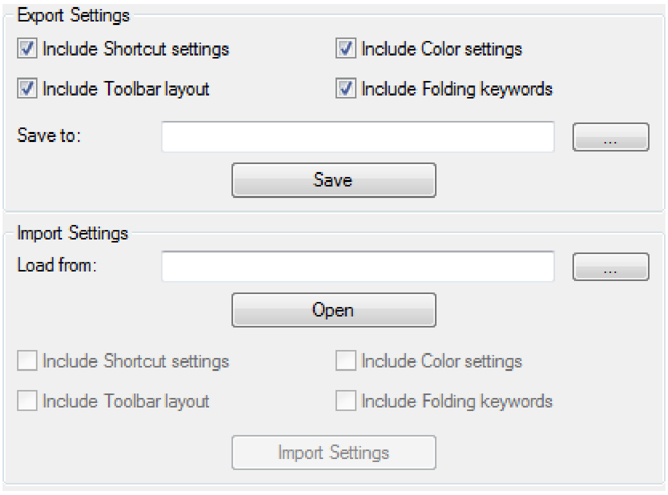
☐ Automatically hide the Panel

Milliseconds delay before hiding the Panel: 2500

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


Export Settings
✓ Include Shortcut settings ✓ Include Color settings
✓ Include Toolbar layout ✓ Include Folding keywords
Save to:

Import Settings
Load from:

☐ Include Shortcut settings ☐ Include Color settings
☐ Include Toolbar layout ☐ Include Folding keywords

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 SpiderBasic 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 17

Command-line options for the IDE

The SpiderBasic 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 SpiderBasic 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 .sb
 extension in the registry
/LOCAL                  puts all preferences in the SpiderBasic
 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%\SpiderBasic\ directory on the system.

The /NOEXT command is useful when you have several different SpiderBasic versions at once (for testing of beta versions for example), but want the .sb 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 SpiderBasic to different computers (or run it from USB sticks for example).
Example:

```
  SpiderBasic.exe Example.sb /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 "*.sb" you can load a whole directory).

Part III

Language Reference

Chapter 18

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, SpiderBasic does not currently have a power-of operator! Use the SpiderBasic 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 SpiderBasic'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, SpiderBasic 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 SpiderBasic 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 SpiderBasic 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 SpiderBasic, you put a dollar sign in front of the number, for example $BEEF.

Chapter 19

Break : Continue

Syntax

```
Break
```

Description

Break provides the ability to exit during any iteration, for the following loops: Repeat , For , ForEach and While .

Example

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

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

Chapter 20

Using the command line compiler

Introduction

The command line compiler is located in the subdirectory 'Compilers\' from the SpiderBasic folder. The easier way to access this is to add this directory in the windows PATH variable, which will give access to all the commands of this directory at all times. The Linux/OS X switches equivalent are also accepted on Windows to ease script creation across all platforms.

Command switches

/? (-h, –help): display a quick help about the compiler.
/COMMENTED (-c, –commented): create a commented javascript output file.
/DEBUGGER (-d, –debugger): enable the debugger support.
/OUTPUT (-o, –output) "filename": output name for the created web, iOS or Android app.
/JAVASCRIPT (-js, –javascript) "filename.js": name of the created javascript file.
/LIBRARYPATH (-lp, –librarypath) "path": path of the SpiderBasic dependencies.
/COPYLIBRARIES (-cl, –copylibraries) "path": copy the SpiderBasic dependencies to the specified local path.
/RESOURCEDIRECTORY (-rd, –resourcedirectory) "path": the directory where all the app resources are located.
/OPTIMIZEJS (-z, –optimizejs): creates an optimized javascript output, including needed javascript. A recent Java JRE needs to be installed to have this option working. The most recent JRE version can be found here: https://java.com/download.
/ICON (-n, –icon) \"filename.png\": icon to display in the browser tab. Has to be in PNG image format.
/APPNAME (-an, –appname): set the app name.
/APPVERSION (-av, –appversion): set the app version.
/APPPERMISSION (-ap, –apppermission) "permission": add a permission to the app (it can be specified several time). Available permissions:
- geolocation: add GPS access and other location API support to the app.
/KEEPAPPDIR (-ka, –keepappdir): don't delete Cordova temp directory after app creation (iOS and Android only)
/REBUILDAPPDIR (-ra, –rebuildappdir) \"path\": rebuild the specified Cordova temp app directory (iOS and Android only)
/STARTUPIMAGE (-si, –startupimage) "image.png": set the startup image to use when launching the app (iOS and Android only)
/PACKAGEID (-pi, –packageid) "com.yourcompany.appid": set the package id for the app (iOS and Android only)
/IAPKEY (-ik, –iapkey) "MIIB......AQAB": set InAppPurchase API key for the Android app
/DEPLOY (-p, –deploy): automatically deploy the app on USB connected device (iOS and Android only)
/FULLSCREEN (-fs, –fullscreen): activate fullscreen mode for the app (iOS and Android only)

/ORIENTATION (-w, –orientation) ”orientation”: set the app orientation (iOS and Android only):
- ”any”: the app can be in landscape or portrait mode, depending of the device orientation (default).
- ”portrait”: the app will always launch in portait mode.
- ”landscape”: the app will always launch in landscape mode.
/ANDROID: create an Android app (Windows only).
/JDK ”jdk path”: set the path to full JDK 1.8+ used to build Android app (Windows only).
/RESIDENT (-r, –resident) ”filename”: create a resident file specified by the filename.
/QUIET (-q, –quiet): Disable all unnecessary text output, very useful when using another editor.
/STANDBY (-sb, –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 SpiderBasic 'SDK' directory.
/IGNORERESIDENT (-ir, –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.
/CONSTANT (-t, –constant) Name=Value: Creates the specified constant with the given expression. Example: 'sbcompiler test.sb /CONSTANT MyConstant=10'. The constant #MyConstant will be created on the fly with a value of 10 before the program gets compiled.
/SUBSYSTEM (-s, –subsystem) ”Name”: Uses the specific subsystem to replace a set of internal functions. For more information, see subsystems .
/CHECK (-k, –check) : Check the syntax only, doesn't create or launch the executable.
/PREPROCESS (-pp, –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 SpiderBasic 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 SpiderBasic source file to create the final output.
/LANGUAGE (-g, –language) ”Language”: uses the specified language for the compiler.
/VERBOSE (-vb, –verbose): Displays more information when creating an app.
/VERSION (-v, –version): Displays the compiler version.

Example

```
  CLI> sbcompiler sourcecode.sb /OUTPUT "test.html"
```

The compiler will compile the source code to test.html.

```
  CLI> sbcompiler sourcecode.sb /DEBUGGER /OUTPUT "test.html"
```

The compiler will compile the source code to test.html with debugger support.

Chapter 21

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

```
  CompilerIf #PB_Compiler_OS = #PB_OS_Linux And #PB_Compiler_Processor
   = #PB_Processor_x86
    ; some Linux and x86 specific code.
  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_AmigaOS
    ; some Amiga specific code
  CompilerCase #PB_OS_Linux
    ; some Linux specific code
CompilerEndSelect
```

Syntax

```
CompilerError <string constant>
```

Description

Generates an error, 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.

Example

```
CompilerIf #PB_Compiler_OS = #PB_OS_AmigaOS
  CompilerError "AmigaOS isn't supported, sorry."
CompilerElse
  CompilerError "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

```
EnableJS
DisableJS
```

Description

Enables or disables the inline javascript. Inside this block, the line are left untouched and put as is in the generated code. See the inline javascript section for more information.

Example

```
;
;
Test = 10

EnableJS
  v_test = 20
DisableJS

Debug Test ; Will be 20
```

Reserved Constants

The SpiderBasic 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 creating Windows executable
 (PureBasic)
  #PB_OS_Linux   : The compiler is creating Linux executable
 (PureBasic)
  #PB_OS_AmigaOS : The compiler is creating AmigaOS executable
 (PureBasic)
  #PB_OS_MacOS   : The compiler is creating OS X executable
 (PureBasic)
  #PB_OS_Web     : The compiler is generating a JavaScript file
 (SpiderBasic)

#PB_Compiler_Processor : Determines the processor type for which the
 output is created. It can be one of the following:
  #PB_Processor_x86     : x86 processor architecture (also called
 IA-32 or x86-32) (PureBasic)
  #PB_Processor_x64     : x86-64 processor architecture (also called
 x64, AMD64 or Intel64) (PureBasic)
  #PB_Processor_PowerPC : PowerPC processor architecture (PureBasic)
  #PB_Processor_mc68000 : Motorola 68000 processor architecture
 (PureBasic)
  #PB_Processor_JavaScript : JavaScript output (SpiderBasic)

#PB_Compiler_App : Determines the app type for which the output is
 created. It can be one of the following:
  #PB_App_Web     : Web app
  #PB_App_Android: Android app
  #PB_App_IOS     : IOS app

#PB_Compiler_Date      : Current date, at the compile time, in the
 SpiderBasic  date
 format.
  #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 SpiderBasic 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 a final project is created
                          is created, the debugger is always disabled
   (this constant will be 0).
  #PB_Compiler_InlineJavaScript : Set to 1 if the if the code is inside
   a EnableJS/DisableJS block.
  #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.
```

Chapter 22

Compiler Functions

Syntax

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

Description

SizeOf can be used to find out the size of any complex Structure (it does not work on the simple built-in types such as word and float), Interface or even variables . This can be useful in many areas such as calculating memory requirements for operations, using API commands, etc.

Example

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

Debug "The size of my friend is "+Str(Sizeof(Person))+" bytes" ; will
 be 10 (4+4+2)

John.Person\Name = "John"

Debug SizeOf(John) ; will be also 10
```

Note: if a variable and a structure have the same name, the structure will have the priority over the variable.

Syntax

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

Description

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

Example

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

Debug OffsetOf(Person\Age) ; will be 8 as a string is 4 byte in memory
                           ; (16 with the 64-bit compiler, as a
 string is 8 bytes there)


Interface ITest
  Create()
  Destroy(Flags)
EndInterface

Debug OffsetOf(ITest\Destroy()) ; will be 4
```

Syntax

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

Description

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

Example

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

If TypeOf(Person\Age) = #PB_Word
  Debug "Age is a 'Word'"
EndIf

Surface.f
If TypeOf(Surface) = #PB_Float
  Debug "Surface is a 'Float'"
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 name of the subsystem is case sensitive.

Example

```
CompilerIf Subsystem("OpenGL")
  Debug "Compiling with the OpenGL subsystem"
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. 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).
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
```

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

```
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 only and should be used with care.

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 free the memory of a structured memory area. This is useful when the structure contains strings, array, list or map which have been allocated internally by SpiderBasic. 'Structure' is 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. This function is for advanced users only and should be used with care.

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

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

Description

Bool can be used to evaluate a boolean expression outside of regular conditional operator like If, While, Until etc. If the boolean expression is true, it will return #True, otherwise it will return #False.

Example

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

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

Chapter 23

Data

Introduction

SpiderBasic 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.

Commands

Syntax

```
DataSection
```

Description

Start a data section.

Syntax

```
EndDataSection
```

Description

End a data section.

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 ','.

Example

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

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

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.

Chapter 24

Debugger keywords in SpiderBasic

Overview

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.

Syntax

```
CallDebugger
```

Description

This invokes the ”debugger” and freezes the program immediately. This will invoke the web browser debugger.

Syntax

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

Description

Display the DebugOutput window and the result inside it. The expression can be any valid SpiderBasic 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.

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.

Chapter 25

Define

Syntax

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

Description

If no <variables> are specified, Define is used to change the default type for future untyped variables (including procedure parameters and interface method parameters). The initial default type is integer (.i). Each variable can have a default value directly assigned to it.
Define may also be used with arrays , lists and maps .

Example

```
d = e + f
Define.w
a = b + c
```

d, e and f will be created as integer type variables, since there was no type specified. a, b and c will be signed word typed (.w) as no type is specified and the default type had been changed to the word type.
If variables are specified, Define only declares these variables as "defined type" and will not change the default type. If you don't assign a value to the variables at this time, the value will be 0.

Example

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

Syntax

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

Description

Alternative possibility for the variable declaration using Define.

Example

```
  Define MyChar.c
  Define MyLong.l
  Define MyWord.w

  Debug SizeOf(MyChar)   ; will print 2
  Debug SizeOf(MyLong)   ; will print 4
  Debug SizeOf(MyWord)   ; will print 2
```

Chapter 26

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 SpiderBasic 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 SpiderBasic (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 .
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

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

Example: Multidimensional array

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

Example: Array as procedure parameter

```
Procedure fill(Array Numbers(1), Length)  ; the 1 stays 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 A() 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 can not 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
```

Chapter 27

Enumerations

Syntax

```
Enumeration [name] [<constant> [Step <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.

Example: Simple enumeration

```
  Enumeration
    #GadgetInfo ; Will be 0
    #GadgetText ; Will be 1
    #GadgetOK   ; Will be 2
  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
```

Chapter 28

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

```
For k = 0 To 10
  Debug k
Next
```

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

Example

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

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

Example

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

```
  For k = a+2 To b+7 Step 2
    Debug k
  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

```
  For x=0 To 10
    For y=0 To 5
      Debug "x: " + x + " y: " + y
    Next y
  Next x
```

Note: Be aware, that in SpiderBasic 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 29

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.
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

```
NewList Number()

AddElement(Number())
Number() = 10

AddElement(Number())
Number() = 20

AddElement(Number())
Number() = 30

ForEach Number()
  Debug Number() ; Will output 10, 20 and 30
Next
```

Example: Map

```
NewMap Country.s()

```

```
Country("US") = "United States"
Country("FR") = "France"
Country("GE") = "Germany"

ForEach Country()
  Debug Country()
Next
```

Chapter 30

General Rules

SpiderBasic has established rules which never change. These are:

Comments

Comments are marked by ;. All text entered after ; is ignored by the compiler.

Example

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

Keywords

All **keywords** are used for general things inside SpiderBasic, 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 for SpiderBasic **functions**.

Example

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

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

Functions

All **functions** must be followed by a (or else it will not be considered as a function, (even for null parameter functions).

Example

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

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

Constants

All constants are preceded by #. 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

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

Labels

All labels must be followed by a : (colon). Label names may not contain any operators (+,-,...) as well special characters (ß,ä,ö,ü,...). When defined in a procedure , the label will be only available in this procedure.

Example

```
  I_am_a_label:
```

Expressions

An expression is something which can be evaluated. An expression can mix any variables, constants, or functions, of the same type. When you use numbers in an expression, you can add the Keyword $ sign in front of it to mean a hexadecimal number, or a Keyword % sign to mean a binary number. Without either of those, the number will be treated as decimal. Strings must be enclosed by inverted commas.

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 : Debug "Not Crazy" : Else : Debug "Crazy" : EndIf
```

Line continuation

If the line contains a big expression, it can be split on several lines. A split line has to end with one of the following operator: 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"

OpenWindow(0,
           10, 10,
           320, 200,
           "Image viewer")
```

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 SpiderBasic documentation are used the terms "commands" and "functions" as well - this is one and the same, independently of a return-value. If a value is returned by the command or function, you can read in the related command description.

Chapter 31

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. 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

```
Global a.l, b.b, c, d = 20

Procedure Change()
  Debug a  ; Will be 10 as the variable is global
EndProcedure

a = 10
Change()
```

Example: With array

```
Global Dim Array(2)

Procedure Change()
  Debug Array(0)  ; Will be 10 as the array is global
EndProcedure

Array(0) = 10
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
```

Chapter 32

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:

```
EventGadget()
EventMenu()
EventWindow()
```

Handles

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

Example

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

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

```
FontID()
GadgetID()
ImageID()
ThreadID()
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 33

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.

Example

```
If a=10
  Debug "a=10"
Else
  Debug "a<>10"
EndIf
```

Example

```
If a=10 And b>=10 Or c=20
  If b=15
    Debug "b=15"
  Else
    Debug "Other possibility"
  EndIf
Else
  Debug "Test failure"
EndIf
```

Chapter 34

Import : EndImport

Syntax

```
Import "cordova command"
EndImport
```

Description

For advanced programmers. Import : EndImport allows to easy invoke a cordova command at build time. It's an Android and iOS functionnality only. The command has to start with 'cordova' followed by any supported parameters.

Example

```
; Will addd the camera plugin to the app, so it can be accessed with
 inline javascript
;
Import "cordova plugin add cordova-plugin-camera"
EndImport
```

Chapter 35

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

```
IncludeFile "Sources\myfile.sb" ; 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

```
XIncludeFile "Sources\myfile.sb" ; This file will be inserted.
XIncludeFile "Sources\myfile.sb" ; 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

```
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

```
IncludePath  "Sources\Data"
IncludeFile  "Sprite.sb"
XIncludeFile "Music.sb"
```

Chapter 36

Inline Javascript

Introduction

SpiderBasic allows you to include raw JavaScript directly into the source code. To activate the inline JavaScript input, you can use the compiler directives EnableJS and DisableJS . Another way is to put '!' (exclamation mark) at the beginning of the line, and the whole line will be put as is in the generated source.
Please note than the JavaScript code isn't checked by the SpiderBasic compiler, so if a syntax error occurs in this code, it will be reported in the browser when executing the code.
The IDE coloring can also be wrong sometimes with JavaScript input, but it won't impact the generated code.

Rules

Here are the naming rules to use when accessing SpiderBasic items: - JavaScript variable name is the same in lowercase with a 'v_' prefix. It's the same for local variable, global variable and function parameter.

Example

```
MyVariable = 10
!v_myvariable++;
Debug MyVariable
```

- JavaScript pointer name is the same in lowercase with a 'p_' prefix. It's the same for local pointer, global pointer and function parameter.

Example

```
*Pointer = 10
!p_pointer = 20;
Debug *Pointer
```

- JavaScript procedure name is the same in lowercase with a 'f_' prefix.

Example

```
Procedure MyProcedure()
  Debug "Hello world"
EndProcedure

; Call in SpiderBasic
MyProcedure()

; Call using Javascript
!f_myprocedure();
```

- JavaScript static variable name in procedure is 'so_procedurename$v_variablename'. Example

```
Procedure myProcedure()
Static myStatic=0

! so_myprocedure$v_mystatic += 12;

Debug myStatic
EndProcedure

myProcedure()
```

- JavaScript module name is the same in lowercase and using the '$' sign to access module items. Example

```
DeclareModule MyModule
  MyModuleVariable.i
EndDeclareModule

Module MyModule
EndModule

!mymodule$v_mymodulevariable = 12
Debug MyModule::MyModuleVariable
```

Chapter 37

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 SpiderBasic, 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 (Theses 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.
Note: The concept of objects, and the capability provided within SpiderBasic 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

```
; In order to access an external object (within a DLL for example),
; the objects' interface must first be declared:

Interface MyObject
  Move(x,y)
  MoveF(x.f,y.f)
  Destroy()
EndInterface

```

```
; CreateObject is the function which creates the object, from the DLL,
; whose interface has just been defined.
; Create the first object...
;
Object1.MyObject = MyCreateObject()

; And the second one.
;
Object2.MyObject = MyCreateObject()

; Then the functions which have just been defined, may
; be used, in order to act upon the desired object.
;
Object1\Move(10, 20)
Object1\Destroy()

Object2\MoveF(10.5, 20.1)
Object2\Destroy()
```

Example: Example with 'Extends'

```
; Define a basic Cube interface object.
;
Interface Cube
  GetPosition()
  SetPosition(x)
  GetWidth()
  SetWidth(Width)
EndInterface

Interface ColoredCube Extends Cube
  GetColor()
  SetColor(Color)
EndInterface

Interface TexturedCube Extends Cube
  GetTexture()
  SetTexture(TextureID)
EndInterface

; The interfaces for 3 different objects have now been defined, these objects include:
;
; - 'Cube' which has the: Get/SetPosition() and Get/SetWidth() functions.
; - 'ColoredCube' which has the: Get/SetPosition(), Get/SetWidth() and Get/SetColor() functions.
; - 'TexturedCube' which has the: Get/SetPosition(), Get/SetWidth() and Get/SetTexture() functions.
;
```

Chapter 38

Licenses for the SpiderBasic applications

All components used by SpiderBasic are under very permissive licenses which allow free use in commercial application. Ensures to put this file in you documentation if you distribute your application. This program makes use of the following components:

Using MIT License

spark-md5 (https://github.com/satazor/js-spark-md5)
js-crc (https://github.com/emn178/js-crc)
js-sha3 (https://github.com/emn178/js-sha3)
canvas-toBlob.js (https://github.com/eligrey/canvas-toBlob.js)
FileSaver.js (https://github.com/eligrey/FileSaver.js)
interact.js (https://github.com/taye/interact.js)
jDataView (https://github.com/jDataView/jDataView)
jquery (https://github.com/jquery/jquery)
jquery-ui (https://github.com/jquery/jquery-ui)
jquery-injectCSS (https://github.com/kajic/jquery-injectCSS)
jquery-blockui (https://github.com/malsup/blockui)
pako (https://github.com/nodeca/pako)
PaperJS (http://paperjs.org)
PixiJS (https://github.com/pixijs/pixijs)
Platform.js (https://github.com/bestiejs/platform.js)
PreloadJS (https://createjs.com/preloadjs)
put-js (https://github.com/put-dev/put-js)
seedrandom (https://github.com/davidbau/seedrandom)
SoundJS (https://www.createjs.com/soundjs)
sql.js (https://sql.js.org)
wgxpath (https://github.com/google/wicked-good-xpath)
xdate (https://github.com/arshaw/xdate)
xregexp (https://github.com/slevithan/xregexp)

Using BSD License

Dojo (https://github.com/dojo/dojo)

Using BSD License

Forge (https://github.com/digitalbazaar/forge)

Using Apache License 2.0

localForage (https://github.com/localForage/localForage)
mouseTrap (https://github.com/ccampbell/mousetrap)

```
   of this License, Derivative Works shall not include works that
remain
   separable from, or merely link (or bind by name) to the
interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or
additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright
owner
   or by an individual or Legal Entity authorized to submit on
behalf of
   the copyright owner. For the purposes of this definition,
"submitted"
   means any form of electronic, verbal, or written communication
sent
   to the Licensor or its representatives, including but not limited
to
   communication on electronic mailing lists, source code control
systems,
   and issue tracking systems that are managed by, or on behalf of,
the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a
Contribution."

   "Contributor" shall mean Licensor and any individual or Legal
Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have
made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
```

```
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute
must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works;
or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The
contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state
otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or
modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the
trade
   names, trademarks, service marks, or product names of the
Licensor,
   except as required for reasonable and customary use in describing
the
```

Chapter 39

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.
The 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). The macros can't be called recursively.

Example: Simple macro

```
Macro MyNot
  Not
EndMacro

a = 0
If MyNot a    ; Here the line will be expanded to : 'If Not a'
  Debug "Ok"
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(Body)
  MessageRequester(UCase(Body), #PB_MessageRequester_Ok)
EndMacro

Text$ = "Hello World"
UMsgBox("-"+Text$+"-") ; Here the line will be expanded like that:
                       ; 'MessageRequester(UCase("-"+Text$+"-"),
 #PB_MessageRequester_Ok)'
```

Example: Macro with default parameter

```
Macro UMsgBox(Body = "Ha, no body specified")
  MessageRequester(UCase(Body), #PB_MessageRequester_Ok)
EndMacro

UMsgBox() ; Here the line will be expanded like that:
          ; 'MessageRequester(UCase("Ha, no body specified"),
 #PB_MessageRequester_Ok)'
```

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

```
Macro Test
  Debug "1"
EndMacro

Test ; Call the macro

UndefineMacro Test ; Undefine the macro, it no more exists

Macro Test ; Now we can redefine the macro
  Debug "2"
EndMacro

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

```
Macro Test
  Debug MacroExpandedCount
EndMacro

Test ; Call the macro
Test ; Call the macro
Test ; Call the macro
```

Chapter 40

Pointers and memory access

Pointers

Using pointers is possible by placing one '*' (asterix) in front of the name of a variable , array , list or map . A pointer is a placeholder for a memory address which is usually associated to a structure .

Example

```
*MyScreen.Screen = OpenScreen(0, 320, 200, 8, 0)
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 SpiderBasic 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 guaranteed address integrity at the compilation time whatever the CPU address mode is.
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

```
Structure Point
```

```
  x.l
  y.l
EndStructure

Define Point1.Point, Point2.Point

*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

```
Structure Point
  x.l
  y.l
EndStructure

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 Arithmetic
In SpiderBasic, a pointer can't be moved by adding or substracting a number. A pointer is only allowed on a structured element. A memory buffer allocated with AllocateMemory() is not allowed.
AllocateStructure() needs to be used if a pointer is a required on a dynamically allocated element.

Address of variable

To find 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 People
  Age.b
  Name$
EndStructure

Procedure SetInfo(*People.People)
  *People\Age = 69
  *People\Name$ = "John"
EndProcedure

Define.People King

SetInfo(@King)

Debug King\Age
Debug King\Name$
```

Address of procedure

The most common reason to get the address of a procedure is when dealing with callbacks. Many commands in SpiderBasic requires callbacks as parameter, like BindEvent() , BindGadgetEvent() , ReadFile() etc. The address of a procedure is got using the 'at' symbol (@) in front of the procedure name including the '()'.

Example

```
Procedure ButtonHandler()
  Debug "Button click event on gadget #" + EventGadget()
EndProcedure

OpenWindow(0, 100, 100, 200, 50, "Click test", #PB_Window_SystemMenu)
  ButtonGadget(0, 10, 10, 180, 30, "Click me")

BindGadgetEvent(0, @ButtonHandler())
```

Chapter 41

Module

Syntax

```
DeclareModule <name>
  ...
EndDeclareModule

Module <name>
  ...
EndModule

UseModule <name>
UnuseModule <name>
```

Description

Modules are an easy way to isolate code part 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 this case, if a module name conflict, the module items won't be imported and a compiler error will be raised. UnuseModule remove 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 modules which needs it. It's the common way have global data available for all modules.
Default items available in modules are all SpiderBasic commands, structure and constants. Therefore module items can not be named like internal SpiderBasic commands, structure 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 SpiderBasic 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

```

  ; Every items in this sections will be available from outside
  ;
  DeclareModule Ferrari
    #FerrariName$ = "458 Italia"
    
    Declare CreateFerrari()
  EndDeclareModule
  
  ; Every items in this sections will be private. Every names can be used without conflict
  ;
  Module Ferrari
    
    Global Initialized = #False
    
    Procedure Init() ; Private init procedure
      If Initialized = #False
        Initialized = #True
        Debug "InitFerrari()"
      EndIf
    EndProcedure
    
    Procedure CreateFerrari()
      Init()
      Debug "CreateFerrari()"
    EndProcedure
    
  EndModule
  
  
  Procedure Init() ; Main init procedure, doesn't conflict with the Ferrari Init() procedure
    Debug "Main init()"
  EndProcedure
  
  Init()
  
  Ferrari::CreateFerrari()
  Debug Ferrari::#FerrariName$
  
  Debug "------------------------------"
  
  UseModule Ferrari ; Now import all public item directly in the main program scope
  
  CreateFerrari()
  Debug #FerrariName$
```

Example: With a common module

```

; The common module, which will be used by others to share data
;
DeclareModule Cars
  Global NbCars = 0
EndDeclareModule

Module Cars
EndModule

; First car module
;
DeclareModule Ferrari
EndDeclareModule

Module Ferrari
  UseModule Cars

  NbCars+1
EndModule

; Second car module
;
DeclareModule Porche
EndDeclareModule

Module Porche
  UseModule Cars

  NbCars+1
EndModule

Debug Cars::NbCars
```

Chapter 42

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

```
NewList MyList()

AddElement(MyList())
MyList() = 10

AddElement(MyList())
MyList() = 20

AddElement(MyList())
MyList() = 30

ForEach MyList()
  Debug MyList()
Next
```

Example: List as procedure parameter

```
NewList Test()

AddElement(Test())
Test() = 1
AddElement(Test())
```

```
Test() = 2

Procedure DebugList(c, List ParameterList())

  AddElement(ParameterList())
  ParameterList() = 3

  ForEach ParameterList()
    MessageRequester("List", Str(ParameterList()))
  Next

EndProcedure

DebugList(10, Test())
```

Chapter 43

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 for an assignment, a new element is automatically added to the map. 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

```
NewMap Country.s()

Country("GE") = "Germany"
Country("FR") = "France"
Country("UK") = "United Kingdom"

Debug Country("FR")

ForEach Country()
  Debug Country()
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
```

Chapter 44

Others Commands

Syntax

```
End [ExitCode]
```

Description

Ends the program execution correctly. If the application is a mobile application, it quits the app. If the app runs in a browser, it will try to close the current window or tab (but can fail depending of the browser).
'ExitCode' is currently ignored.

Syntax

```
Swap <expression>, <expression>
```

Description

Swaps the value of the both expression, in an optimized way. The both <expression> have to be a variable , array , list or a map element (structured or not) and have to be one of the SpiderBasic 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 45

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 it's own variables . In SpiderBasic, 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 them by using Shared or Global keywords (see also the Protected and Static keywords).
The last parameters can have a default value (need 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 returned value will be undefined.

Example: Procedure with a numeric variable as return-value

```
Procedure Maximum(nb1, nb2)
  If nb1 > nb2
    Result = nb1
  Else
    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("SpiderBasic", "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))
```

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

```
Declare Maximum(Value1, Value2)

Procedure Operate(Value)
  Maximum(10, 2)      ; At this time, Maximum() is unknown.
EndProcedure

Procedure Maximum(Value1, Value2)
  ProcedureReturn 0
EndProcedure
```

Chapter 46

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

```
Global a
a = 10

Procedure Change()
  Protected a
  a = 20
EndProcedure

Change()
Debug a ; Will print 10, as the variable has been protected.
```

Example: With array

```
Global Dim Array(2)
Array(0) = 10

Procedure Change()
  Protected Dim Array(2) ; This array is protected, it will be local.
  Array(0) = 20
EndProcedure

Change()
```

```
  Debug Array(0) ; Will print 10, as the array has been protected.
```

Chapter 47

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).
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.

Example

```
Prototype ProtoMax(Value, Print = #False)

Procedure Max1(Value, Print)
  Debug "Max1 : " + Value
EndProcedure

Procedure Max2(Value, Print)
  Debug "Max2 : " + Value
EndProcedure

Max.ProtoMax = @Max1() ; Set the function pointer
Max(10)

Max.ProtoMax = @Max2() ; Change the function pointer
Max(20) ; the function call doesn't change
```

Chapter 48

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

```
a = 0
Repeat
  a = a+1
  Debug a
Until a > 10
```

This will loop until "a" takes a value > to 10, (it will loop 11 times).

Chapter 49

Residents

Description

Residents are precompiled files which are loaded when the compiler starts. They can be found in the 'residents' folder of the SpiderBasic installation path. A resident file must have the extension '.res' and can contain the following items: structures , interfaces , 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 'SpiderBasic.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 SpiderBasic 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 (-ir or –ignoreresident) and /RESIDENT (-r or –resident) 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 50

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.
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

```
Runtime Procedure OnEvent()
  Debug "OnEvent"
EndProcedure

Debug GetRuntimeInteger("OnEvent()") ; Will display the procedure address
```

Example: Enumeration

```
Runtime Enumeration
  #Constant1 = 10
  #Constant2
  #Constant3
EndEnumeration

Debug GetRuntimeInteger("#Constant1")
Debug GetRuntimeInteger("#Constant2")
Debug GetRuntimeInteger("#Constant3")
```

Example: Variable

```
Define a = 20
Runtime a

Debug GetRuntimeInteger("a")
SetRuntimeInteger("a", 30)

Debug a ; the variable has been modified
```

Chapter 51

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). 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

```
  Value = 2

  Select Value
    Case 1
      Debug "Value = 1"

    Case 2
      Debug "Value = 2"

    Case 20
      Debug "Value = 20"

    Default
      Debug "I don't know"
  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 52

Using several SpiderBasic versions on Windows

Overview

It is possible to install several SpiderBasic versions on your hard disk at the same time. This is useful to finish one project with an older SpiderBasic version, and start developing a new project with a new SpiderBasic version.

How to do it

Create different folders like "SpiderBasic_1.00" and "SpiderBasic_1.10" and install the related SpiderBasic version in each folders.
When one "SpiderBasic.exe" is started, it will assign all ".sb" files with this version of SpiderBasic. So when a source code is loaded by double-clicking on the related file, the currently assigned SpiderBasic version will be started. Beside SpiderBasic will change nothing, which can affect other SpiderBasic versions in different folders.
To avoid the automatic assignment of ".sb" files when starting the IDE, a shortcut can be created from SpiderBasic.exe with "/NOEXT" as parameter. The command line options for the IDE are described here .
Note: The settings for the IDE are saved in the %APPDATA%\SpiderBasic 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 SpiderBasic directory and disabled the creation of the .sb extension.

Chapter 53

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 54

SpiderBasic objects

Introduction

The purpose of this section is to describe the behavior, creation, and handling of objects in SpiderBasic. For the demonstration, we will use the Image object, but the same logic applies to all other SpiderBasic 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

```
  CreateImage(0, 640, 480) ; Create an image, the n°0
  ResizeImage(0, 320, 240) ; Resize the n°0 image
```

Example

```
  CreateImage(2, 640, 480) ; Create an image, the n°2
  ResizeImage(2, 320, 240) ; Resize the n°2 image
```

```
CreateImage(2, 800, 800) ; Create a new image in the n°2 index, the old one is automatically free'ed
```

Example

```
For k = 0 To 9
  CreateImage(k, 640, 480) ; Create 10 different images, numbered from 0 to 9
  ResizeImage(k, 320, 240) ; Create a new image in the n°2 index, the old one is automatically free'ed
Next
```

Example

```
#ImageBackground = 0
#ImageButton     = 1

CreateImage(#ImageBackground, 640, 480) ; Create an image (n°0)
ResizeImage(#ImageBackground, 320, 240) ; Resize the background image
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. SpiderBasic 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

```
DynamicImage1 = CreateImage(#PB_Any, 640, 480) ; Create a dynamic image
ResizeImage(DynamicImage1, 320, 240) ; Resize the DynamicImage1
```

Overview of the different SpiderBasic objects

Different SpiderBasic 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 SpiderBasic differs them by their type:

- Font
- Gadget
- JSON
- Image

- Menu (not MenuItem() , as this is no object)
- RegularExpression
- Sound
- Sprite
- ToolBar
- Window
- XML

Chapter 55

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

```
Global a
a = 10

Procedure Change()
  Static a
  a+1
  Debug "In Procedure: "+Str(a) ; Will print 1, 2, 3 as the variable increments at each procedure call.
EndProcedure

Change()
Change()
Change()
Debug a ; Will print 10, as the static variable doesn't affect global one.
```

Example: With array

```
Global Dim Array(2)
Array(0) = 10
```

```

Procedure Change()
  Static Dim Array(2)
  Array(0)+1
  Debug "In Procedure: " + Array(0) ; Will print 1, 2, 3 as the value
 of the array field increments at each procedure call.
EndProcedure

Change()
Change()
Change()
Debug Array(0) ; Will print 10, as the static array doesn't affect
 global one.
```

Chapter 56

Structures

Syntax

```
Structure <name> [Extends <name>]
  ...
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. Statics arrays are supported inside structures.
Dynamic objects like arrays, lists and maps are 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.
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.
SizeOf can be used with structures to get the size of the structure and OffsetOf can be used to retrieve the index 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++/JavaScript 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.
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 SpiderBasic 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.
It's possible to perform a full structure copy by using the equal affectation between two structure element of the same type.
ClearStructure can be used to clear a structured memory area. It's for advanced use only, when pointers are involved.

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: 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

Debug Timo\Next\Name$ ; Will print 'Fred'
```

Chapter 57

Subsystems

Introduction

SpiderBasic integrated commandset relies on specific libraries. Sometimes, there is different way to achieve the same goal and when it makes sense, SpiderBasic offers the possibility to change the used underlying libraries for specific commands, without changing one line of source code.
The available subsystems are located in the SpiderBasic '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

For now, SpiderBasic doesn't have any additional subsystems.

Chapter 58

Variables and Types

Variables declaration

To declare a variable in SpiderBasic, simply type its name. You can also specify the type you want this variable to be. 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

```
  a.b         ; Declare a variable called 'a' from byte (.b) type.
  c.l = a*d.w ; 'd' is declared here within the expression !
```

Notes:

Variable names must not start with a number (0,1,...), contain operators (+,-,...) or special characters (ß,ä,ö,ü,...).
The variables in SpiderBasic 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 SpiderBasic 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

SpiderBasic 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: SpiderBasic 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 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.

```
a.s = "One string"
a$ = "Another string"
Debug a   ; will give "One string"
Debug a$  ; will give "Another string"
```

Note: The floating numbers (floats + doubles) can also be written like this: 123.5e-20

```
value.d = 123.5e-20
Debug value   ; will give 0.000000000000000001235
```

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 SpiderBasic, 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

```
count=total/price ; Divides the value of "total" by the value of
  "price" and uses the result with the equals operator
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

```
; Shown using binary numbers as it will be easier to see the result
a.w = %1000 & %0101 ; Result will be 0
b.w = %1100 & %1010 ; Result will be %1000
bits = a & b ; AND each bit of a and b and use result in equals
  operator
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

```
; Shown using binary numbers as it will be easier to see the result
a.w = %1000 | %0101 ; Result will be %1101
b.w = %1100 | %1010 ; Result will be %1110
bits = a | b ; OR each bit of a and b and use result in equals
  operator
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

```
; Shown using binary numbers as it will be easier to see the result
a.w = %1000 ! %0101 ; Result will be %1101
b.w = %1100 ! %1010 ; Result will be %0110
bits = a ! b ; XOR each bit of a and b and use result in equals
  operator
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

```
; Shown using binary numbers as it will be easier to see the result
a.w = ~%1000 ; Result will be %0111
b.w = ~%1010 ; Result will be %0101
```

Operator () (Brackets)

You can use sets of brackets to force part of an expression to be evaluated first, or in a certain order.

Example

```
a = (5 + 6) * 3 ; Result will be 33 since the 5+6 is evaluated first
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

```
a=%1011 << 1 ; The value of a will be %10110. %1011=11, %10110=22
b=%111 << 4 ; The value of b will be %1110000. %111=7, %1110000=112
c.l=$8000000 << 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

```
d=16 >> 1 ; The value of d will be 8. 16=%10000, 8=%1000
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)
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

```
a=16 % 2 ; The value of a will be 0 as 16/2 = 8 (no remainder)
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

```
Value + 1  ; The same as: Value = Value + 1
Value * 2  ; The same as: Value = Value * 2
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

```
Dim MyArray(10)
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)  |      ~, -
     7         |    <<, >>, %, !
     6         |       |, &
     5         |       *, /
     4         |       +, -
     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 SpiderBasic.
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 59

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

```
b = 0
a = 10
While a = 10
  b = b+1
  If b=10
    a=11
  EndIf
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 60

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 anti-slash '\' character which does not 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

```
Structure Person
  Name$
  Age.l
  Size.l
EndStructure

Friend.Person

With Friend
  \Name$ = "Yann"
  \Age   = 30
  \Size  = 196

  Debug \Size+\Size
EndWith
```

Example: Complex example

```
Structure Body
  Weight.l
  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
```