Superbase NG IDE Users Guide

Building Projects in the SIMPOL Language

Manuel Franco
Neil Robinson
Duncan Jones

Superbase NG IDE Users Guide: Building Projects in the SIMPOL Language

by Manuel Franco, Neil Robinson, and Duncan Jones

Table of Contents

List of Tables

List of Examples

Chapter 1. Copyright and Disclaimer

Copyright Information

This document is copyrighted © 2003-2016 Superbase Software Limited and is not permitted to be distributed by anyone other than Superbase Software Limited and its licencees.

All translations, derivative works, or aggregate works incorporating any of the information in this document must be cleared with the copyright holder except as provided for under normal copyright law.

If you have any questions, please contact `<info@simpol.com>`

Disclaimer

No liability for the contents of this document can be accepted. Use the concepts, examples and other content at your own risk.

All copyrights are held by their respective owners, unless specifically noted otherwise. Use of a term in this document should not be regarded as affecting the validity of any trademark or service mark.

Naming of particular products or brands should not be seen as endorsements.

You are strongly recommended to take a backup of your system before a major installation and backups at regular intervals.

Chapter 2. Introduction

This application is an Integrated Development Environment (IDE) to write, execute and debug Superbase NG applications. There is only one Superbase NG IDE release version, one that runs on Windows XP and later operating systems. The older releases also included a version for Windows 9x operating systems. This is no longer the case.

The application is mainly an editor to write documents in the SIMPOL language. This editor provides a color-coding engine that makes it very easy to program in SIMPOL. There are many features that help the user to write code. For example, a *Find in files* tool, copy and paste, full screen view, and most recently-used file and project lists. It also supports working with different languages in the same Superbase NG application in that other languages can be correctly color-coded, such as XML, HTML, JavaScript, and others. The editor is very flexible and can be easily personalized. The editor supports other languages as XML, HTML, JavaScript, Visual Basic, ….

The objective is to write a complex Superbase NG application quickly and be able to maintain it easily. The IDE manages the dependences among Superbase NG files and compiles, executes and allows debugging of Superbase NG projects, even CGI projects! It also provides project documentation tools to easily document the various components of a project.

Chapter 3. The Superbase NG Project

This chapter briefly covers what a Superbase NG project is and what are its components.

Introduction

Source code files in SIMPOL are stored with one of two file extensions: `sma` or `smu`. The first extension indicates that the file content is stored in an ASCII format (1 byte per character) and the second one indicates that the file content is stored in Unicode format. Unicode can be stored in a number of different formats. SIMPOL Unicode source files currently must be stored in UCS-2 and should begin with what is known as a byte-order mark (BOM). UCS-2 format stores characters using two bytes per character. It is considered good form to also use the byte order mark (0xFEFF) as the first character. This allows the reading program to determine whether the characters are stored with the least significant byte or with the most significant byte first.

Here is an example of SIMPOL source code:

```
function main()
  string s
  s = "Hello"
end function s
```

After compiling a SIMPOL source code file the result is a byte-code file. SIMPOL byte-code files come in two flavors: programs and libraries. These are distinguished by the file extensions `smp` and `sml` respectively. The only difference between them is that the program files are produced from projects that contain a `main()` function. That is the entry point for a SIMPOL program. Compiled programs that do not include a `main()` function can not be executed but can be linked to other programs to provide functions and data types that can be called or used. They can also be loaded dynamically.

Here are two examples of the compilation process:

1. `MySIMPOLFile1.sma` —> `MySIMPOLFile1.smp`

2. `MySIMPOLFile2.sma` —> `MySIMPOLFile2.sml`

After the compilation process, if multiple source files are used to produce the resulting program, the `smp` file is joined with each of the `sml` files in a linking type of process in order to produce the final program.

Here is an example of the linking process:

1. `MySIMPOLFile1.smp` + `MySIMPOLFile2.sml` —> `MyProgram.smp`

Following the linking process, we can execute the SIMPOL program file in the IDE or depending on the type of program from the command line or as the result of entering a URL in a web browser. The IDE will call the SIMPOL virtual machine (SVM)and pass it the program and any command line arguments that have been defined. The SVM then executes the program file. SIMPOL programs usually return a result string, which will be displayed in the IDE or if called from the command line will be sent to standard out. In the case of a web server program, the result is normally a web page.

A SIMPOL project is a group of `sma` and/or `smu` files and the description of how the compilation and link is to be done. It also includes a definition of which directories to search in for included files and potentially one or more targets to be created from the final result. It also includes the list of precompiled libraries to link with, in addition to any library modules that are produced as part of the project itself. SIMPOL source code files can also include any number of other SIMPOL source code files which themselves may include yet others. Typically a project may consist of a main source code

file that then includes other source code files, which may then include others. This results in a tree of files below the main file and this is shown in the project window to the left of the area where the source files are edited. For each module in the project, there is a main source file. Each main source code file is the top of a tree of included source code files. The SIMPOL language statement to include a file is `include` followed by the file name as a string.

It is not possible to simply compile and execute any source code file (if it has a `main()` function) using the IDE. Changes that were made to dynamically load the required components mean that the component information is added to the project by the IDE *build* process. Whether for a simple or complex project, the Superbase NG IDE's real strength is in compiling and linking complex projects. The normal approach to working with the IDE is to create a project and then to build it. This results in either a program or library that can be either executed or loaded into the SVM.

The following is an example of building a project:

1. Project's main source code files: `MyFile1.sma`, `MyFile2.sma`

2. `MyFile1.sma` — includes —> `MyFile1a.sma`, `MyFile1b.sma`

3. `MyFile2.sma` — includes —> `MyFile2a.sma`

4. `MyFile1.sma` — compiles to —> `MyFile1.smp`

5. `MyFile2.sma` — compiles to —> `MyFile2.sml`

6. `MyFile1.smp` + `MyFile2.sml` — links to —> `MyFile1.smp` located in the `bin` directory of the project.

The SIMPOL IDE manages the time dependencies between SIMPOL files, so if in the previous example, we update the file `MyFile2a.sma`, the only file that is going to be compiled when we do a build, is `MyFile2.sma`, because it is the only main source code file affected.

The project description is stored in a file with the extension `smj`. For example, in the previous example the project file name would have been `MyFile2.smj`.

The Organization of Files in a Project

A SIMPOL project is stored on the hard disk as a group of files with the following extensions `smj`, `sma`, `smu`, `smp`, `sml`, and `smz` with an appropiate folder structure. I will explain the folder structure using the above project example.

The basis of every project is a directory. All of the files that are directly part of the project are stored in the project directory or in subdirectories below that. The name of the directory is the same as the name of the project. If the project name is `MyFile1`, the directory name will be `MyFile1`. The project description is stored in this directory; in this example it is named `MyFile1.smj`. Inside the project directory we will have a directory for each main source code file. These are called modules or module directories. Each module contains its main source code file and the rest of the source code files that are implemented as part of the module. Any source code files that are in the module directory besides the main source code file of the module, must be included in the main or other included source code files in this module in order to be compiled and considered part of the module during the build process. It is also possible to include, as part of the code of the module, files from other modules or other projects, for example source code files with standard functions, etc. When the project is built, the resulting byte-code file (either library or program) will also be found in the module's directory.

The name of the module directory will be the name of the main source code file without the extension. In the previous example we have two modules: "MyFile1" and "MyFile2". The first module "My-File1" contains the files `MyFile1a.sma`, `MyFile1b.sma` and after it is compiled the first time, `MyFile1.smp`. The second module "MyFile2" contains the files `MyFile2.sma` and after it has been compiled the first time, `MyFile2.sml`".

When the project is built the result is stored in the directory called `bin`, which is a subdirectory of the project directory. In the example, the final result of the building the project is the file `MyFile1.smp`.

SIMPOL Server Pages

Description

A SIMPOL server page is a file with the extension `smz`, the contents of which is HTML but which also includes blocks of SIMPOL source code. The blocks of source code are inside server page comment blocks (in between `<%` and `%>` tags), so if we launch the HTML viewer component of the SIMPOL IDE, we will see just the HTML page as if it did not contain any SIMPOL source code.

A project with `smz` files is a *CGI project*. This means that the final SIMPOL program is intended to be executed on the server side as a CGI, ISAPI or Fast-CGI program. Typically, a CGI program is called from a web server, for example the Apache web server. The entry point of a CGI or ISAPI program is the `main()` function, but in this case, the function has only one argument: **function main(cgicall cgi)**. `cgi` is a cgicall object, and it contains all of the information the web server received from a browser call.

Building a CGI project adds another process to the normal build. First, the `smz` files are compiled into `sma` or `smu` files, and then the normal build follows. When a SIMPOL server page is compiled into SIMPOL source code, the SIMPOL source code in the server page block comments are passed through without change and the HTML is converted into string arguments of `cgi.output()` statements.

Here is an example of the process followed when building a CGI project:

1. Project's main server page files: `MySPFile1.smz`, `MySPFile2.smz`

2. `MySPFile1.smz` — includes —> `MySPFile1a.txt`

3. `MySPFile1.smz` — compiles to —> `MySPFile1.sma`

4. `MySPFile2.smz` — compiles to —> `MySPFile2.sma`

5. Project's main source code files: `MyFile1.sma`, `MyFile2.sma`

6. `MyFile1.sma` — includes —> `MyFile1a.sma`, `MyFile1b.sma` and `MySPFile1.sma`

7. `MyFile2.sma` — includes —> `MyFile2a.sma` and `MySPFile2.sma`

8. `MyFile1.sma` — compiles to —> `MyFile1.smp`

9. `MyFile2.sma` — compiles to —> `MyFile2.sml`

10. `MyFile1.smp` + `MyFile2.sml` — link to —> `MyFile1.smp`

The following is an example of SIMPOL server page code:

Example 3.1. SIMPOL Server Page Code

```
<%'------------------------- begin code -------------------------
function ShowHelloOrNothing(cgicall cgi, integer i)
'-------------------------  end code  -------------------------%>
<HTML>
 <HEAD><META http-equiv="pragma"
   content="no-cache"></HEAD>
 <TITLE>SIMPOL Hello Page</TITLE>
```

```
<%'------------------------ begin code -------------------------
 if(i == 1)
'------------------------ end code   -------------------------%>
  <BODY>Hello</BODY>
<%'------------------------ begin code -------------------------
 end if
'------------------------ end code   -------------------------%>
</HTML>
<%'------------------------ begin code -------------------------
end function
```

Results after compiling the server page:

Example 3.2. Compiled SIMPOL Server Page

```
'------------------------ begin code -------------------------
function ShowHelloOrNothing(CGICall cgi, integer i)
'------------------------ end code   -------------------------
 cgi.output("<HTML>" + "{0D}{0A}", 1)
 cgi.output(" <HEAD><META http-equiv=""pragma""" + "{0D}{0A}", 1)
 cgi.output("    content=""no-cache""></HEAD>" + "{0D}{0A}", 1)
 cgi.output(" <TITLE>SIMPOL Hello Page</TITLE>" + "{0D}{0A}", 1)
'------------------------ begin code -------------------------
 if(i == 1)
'------------------------ end code   -------------------------
 cgi.output("  <BODY>Hello</BODY>" + "{0D}{0A}", 1)
'------------------------ begin code -------------------------
 end if
'------------------------ end code   -------------------------
 cgi.output("</HTML>" + "{0D}{0A}", 1)
'------------------------ begin code -------------------------
end function
'------------------------ end code   -------------------------
```

The advantage of doing this, is that we can create HTML in a dynamic way using the power of the SIMPOL programming language and we can also visualize the HTML in the HTML viewer that is part of the IDE whenever we need it. So it is very easy to embed HTML (what a final user will see in his browser) in a SIMPOL CGI-style program.

The way that SIMPOL server pages work is different to that of ASP, JSP, or PHP. In each of these cases, the source code is also embedded into the HTML but unlike with SIMPOL these mixed-mode pages are then interpreted by the web server (which must be especially designed to be aware of them) and then the code portions are passed to the language interpreters for execution. With SIMPOL server pages, the design style is similar but the results are compiled rather than interpreted, which is faster and also does not require any special capabilities on the part of the web server.

A CGI project can contain any number of server pages. The server pages follow the same pattern as SIMPOL source code files when including files. Each main server page is the root node of a tree of other included files. The Superbase NG IDE manages the file time-dependencies when a build is done, as in the case of source code SIMPOL files.

The Superbase NG IDE provides a way to compile SIMPOL server pages into SIMPOL source code, and a way to regenerate a SIMPOL server page after manipulating the associated (compiled) SIMPOL source code; this allows the programmer to use the color coding capabilities of the IDE for the HTML source when working on the server page and then after compiling, it is possible to work on the SIMPOL program source in the resulting compiled page. After the changes are done to the compiled source, the option to regenerate the server page from the right mouse button popup menu should be used to send the changes back to the server page source code.

> **Tip**
>
> It is always a good idea to propogate the changes back to the server page source right after making them, since the server page is the reference source code in the project. If you forget, during the next build you will be prompted that the code has changed. If you say okay, your changes will be lost as the compilation of the server page overwrites them. Also, only change code in the blocks between the begin code and end code comments. If you change anything else, it won't successfully regenerate the server page!

Server Page Directives

This section covers the syntax of the server page and the specific directives provided.

Multiline Comments

A multiline comment can consist of any piece of text between the start tag `<%--` and the end tag `--%>`. The comment can cover multiple lines and since it is a comment in the server page it will not be transferred to the source code when that is compiled. Only white space (spaces and tab characters) may precede the begin comment tag on the same line, and only white space may follow the end comment tag on the same line.

Server Page Comment Blocks

These comment blocks are similar to the previous type, except that the contents are passed through to the resulting compiled SIMPOL source code file as source code statements. This is the method by which the embedded source code is extracted into the target program source file. Any text between the start tag `<%` and the end tag `%>` will be transferred as SIMPOL source code to the target source code file. As in the previous case, the start tag may only be preceded on the same line by white space and the end tag may only be followed on the same line by white space.

include

Include the content of a file when the server page is compiled. Example:

```
<%@ include = ".\folder\MyHTMLChunk.txt" %>
```

outputcall

By default a line of HTML code in the server page file is converted into a line in the SIMPOL source code file after compilation. The HTML text is embedded in a SIMPOL string that is an argument to the output method of the cgicall object. It means that if we have a large server page file, after compilation, lots of **output** calls are generated. We can optimize this using the **outputcall = chunk** directive. It can be located at any line in the server page file. To reverse this behaviour we have to use the **outputcall = line** directive. For example:

```
<%@ outputcall = chunk %>
<%@ outputcall = line %>
```

SIMPOL Source Code in an HTML Argument Value

In a line of a server page that holds an HTML argument value in between double quotes, we can add a small piece of SIMPOL source code in between back tick (`` ` ``) character marks. For example:

```
<A href="`sVar1`">Hello</A>
```

Typically it is used to embed a SIMPOL string variable. In this example, after compilation we will get something like this:

```
cgi.output("<A href=""" + sVar1 + """>Hello</A>" + "{0D}{0A}", 1)
```

So, we can see that the value of the HTML argument that the browser will receive is the value of the SIMPOL variable. It is also possible to embed short chunks of inline code.

Chapter 4. The Superbase NG IDE Environment

This chapter covers the various components that make up the Superbase NG Integrated Development Environment (IDE).

Starting the IDE

The IDE can be started from the command line or another program in various ways that will be outlined below:

Command Line	Parameters	Description
sbngide.exe	None	The application is launched.
sbngide.exe	`filename`	First, the application is launched. After that, if the file is a project description file (*`smj`), the project will be loaded into the application. If the file is of any other type of file, it will be opened in the application editor.
sbngide.exe	`/o filename.smj`	This is intended to be used to directly load a project description file (*`smj`). The application is launched and the project is loaded.
sbngide.exe	`/b filename.smj`	This is intended to be used to directly load a project description file (*`smj`). The application is launched and the project is loaded. It then builds the project and the application closes again. Any output from the build process is directed to the shell.
sbngide.exe	`/r filename.smj`	This is intended to be used to directly load a project description file (*`smj`). The application is launched and the project is loaded. It then rebuilds the project and the application closes again. Any output from the rebuild process is directed to the shell.
sbngide.exe	`/e filename.smj`	This is intended to be used to directly load a project description file (*`smj`). The application is launched and the project is loaded. It then executes the project and the application closes again. Any output from the program execution is directed to the shell.
sbngide.exe	`/x filename.smj outfile.xml`	This is intended to be used to directly load a project description file (*`smj`). The application is launched and the project is loaded. It then produces the project information as an XML file and saves it to the filename passed in the second argument and then the application closes again. Any output from the documentation generation process is directed to the shell.
sbngide.exe	`/d filename.smp`	This is used for callback debugging purposes primarily associated with CGI debugging. The IDE is launched and the associated project for the byte-code file is loaded. The program is then

Command Line	Parameters	Description
		placed into debug mode with a break at the first code statement inside the function `main()`.

Editing Documents

Editing documents is the primary objective of the Superbase NG IDE. As in any editor, a user can create a document and store it in a file, open an existing file, update the content before saving it, etc. There are many features in the IDE that make it easy to write and debug SIMPOL program code. Also because it is quite common today to need to work in several different languages, the IDE supports basic color coding for a number of languages, including Microsoft's Visual Basic, Visual Basic Script, JScript, and C#. Also supported are HTML, XML XSL, IDL (Interface Description Language — used for defining CORBA interfaces) and of special interest to Superbase programmers, it supports both tokenized and text format Superbase programs. This can also be extended by the user as needed simply by creating a configuration file based on one of those supplied and then adding it to the list of supported languages. A language is associated with a list of file extensions, so if we open in the editor a file with an extension associated with a language, the editor will apply the color-coding syntax rules to the document. That will show the text in the document coloring keywords, operators, etc, which greatly enhances the ability to read it accurately. The language settings can be personalized through the language settings dialog box.

The primary language used in the Superbase NG IDE is of course the SIMPOL language. There are many built-in features in the editor to handle the specific SIMPOL syntax.

The Help Valet

Many people are familiar with a technology popularized by Microsoft known as IntelliSense®. The Superbase NG IDE has a comparable technology specifically tailored to the needs of the SIMPOL programmer that we call the Help Valet. It is activated whenever there is a project loaded in the IDE and the active document is a SIMPOL document that *belongs* to the project. In this case the editor will take the information that the IDE retrieves from the project, in order to make it easier for the user to understand their own program. There are several Help Valet features, including: data type help, function prototype help, OnMouseOver help, and language items help. Some of the items will not provide complete functionality until after the program has been successfully built at least once. For example, it is not possible to show the members of a user-defined type until that type has been part of a build cycle of the project being edited. The same is true of user-defined functions.

Data Type Help

This is activated whenever the we append the SIMPOL property (dot) operator after an object name in the code. A list is then displayed with the properties of the object. We can use the up and down arrow keys to move through the list, or the mouse cursor to select another list property name. Another way is to begin to write the name of the property so that the property name closest to what has been written will be shown selected in the list. If we press the tab key on the keyboard, the whole property name will be appended after the dot operator.

This feature works with types nested at any level within other types. See the example below:

Example 4.1. Data Type Help

```
type MyType
  embed
  string s1
  integer i1
end type
```

```
function main()
  MyType t

  t.                                                    ❶

end function "OK"
```

❶   After pressing the dot key, a list will display the s1 and i1 property names.

Function Prototype Help

Function prototype help is activated when the open parenthesis character is appended after a function name in the code. A list is then displayed with the parameters of the function. Each entry in the list shows the parameter data type, its name, and even its default value (if it has one). As we add parameters to the function, the list entry selected will be moved one position down, so that the parameter that is selected in the list is the same as the one that we are currently typing. The list will be closed when we type the close parenthesis, that means that the function is not going to receive any more parameters. If the parameter list is still active, we can use the the left and right arrow keys to move to another parameter position. The parameter selected in the list will be the parameter the caret is over in that moment. If we are in a function and one of the parameters we type is another function call, the editor will show the new function parameter list, and after typing all the parameters the new function needs, the editor will show the previous function parameter list in order to continue to support the entering of parameters.

There is another way this feature can be used. If we set the cursor at any position inside a function parameter list in our program and press at the same time the keys Ctrl+Tab, a list with the parameters of the function will be displayed. The parameter entry selected will be the parameter that the cursor points to in the text. This can be very useful if there is a line in the program with many nested functions. For example:

Example 4.2. Function Prototype Help

```
function MyFunction(string s1, integer i1)
end function "OK"

function main()

    MyFunction(                                          ❶

end function "OK"
```

❶   After pressing the open parenthesis key, a list will display the s1 and i1 parameters.

OnMouseOver Help

This functionality is always active. If the mouse cursor is positioned over an item in the function body of the source code, a tooltip will be shown, containing information relevant to the item below the mouse pointer if the item is a function, a type, or a variable. It is a very powerful feature and when debugging the SIMPOL program, if the mouse cursor is positioned over a variable, the value of the variable will be shown.

Example 4.3. OnMouseOver Variable Contents Help (Debugging)

```
function main()
    string s1, s2

    s1 = "Hello"
```

```
    s2 = s1                                                    ❶

end function "OK"
```

❶   If we move the mouse cursor over the `s1` variable in this line, a tooltip will be displayed showing: `string s1 = "Hello"`.

Example 4.4. OnMouseOver Function Prototype Help

```
function MyFunction(string sArg, integer iArg)
end function "OK"


function main()
    string s

    s = MyFunction("Hi", 1)                                    ❶

end function
```

❶   If we move the cursor over the function `MyFunction` in this line, then a tooltip will be displayed showing: `MyFunction(string sArg, integer iArg)`.

Language Items Help

A list is displayed of either functions or types at the current cursor position when the appropiate keys are pressed.

List Type	Key Commands	Description
Intrinsic types	**Ctrl**+**F7**	Shows the SIMPOL language internal types.
User-defined types	**Ctrl**+**Shift**+**F7**	Shows the types specific to the project.
Internal functions	**Ctrl**+**F8**	Shows the internal functions of the SIMPOL language.
User-defined functions	**Ctrl**+**Shift**+**F8**	Shows the functions specific to the project.

Control Bars

Control bars are a set of windows that share a common level of functionality. At the bottom of any control bar there is a tab control that allows easy selection of any window owned by the control bar. There are four different control bars in the Superbase NG IDE: the Project Space control bar, the Output Windows control bar, the Call Stack control bar, and the Variables control bar. The Project Space control bar is the most complex and that will be covered first.

The Project Space Control Bar

This control bar has two windows. They show information for the project that is currently loaded into the IDE. The content of both windows is updated after any project is built.

Project View

In this window the files that make up a project are shown as a tree. The root node is always the project node. The name of the node is the name of the file that contains the information for the project. The extension of the project file is always `smj`, for example: `MyProject.smj`.

The child nodes are the modules. There is one module node for each module directory in the project file structure. The name of the module node is the module directory name. There are two types of modules,

project modules and imported modules. Project modules are modules that belong to the project and that are built as part of the process of building the project. Imported modules are modules that belong to other projects. When we add a module to the project a new module directory is created in the project file structure and a new main source code file is created for the module. When we import a module, what the IDE does is add a link to a module that is located in another project.

In the example used in the previous chapter, we had two module nodes: `MyFile1` and `MyFile2`. A module node always has a child node, which is the main source code node. The name of this node is the same as the name of the main source code node file. This node will have as many child nodes as it has included files. And each of the child nodes will have as many child nodes as they have included files and so on.

In a CGI project we will have the `Server Pages` node as a child node of each module. This node looks like a folder and it will contain all the server pages of the module. The Server Page nodes have the name of the server page file (a file with a `smz` extension). Each Server Page node will have as many child nodes as there are files included in each server page and so on as in the case of the SIMPOL source code files. Server page child nodes are normally files with any extension and that contain chunks of html code.

Project Tree View Nodes

Double-clicking on a source code node or server page node, causes the associated file to be opened in the editor. Right mouse button clicks on any node displays a menu of options specific to the type of node that was clicked on.

The Project Node

This is the root node and it represents the entire project.

Table 4.1. Menu Options

Menu Item	Description
Add New Module	Opens a dialog box to create a module in the active project.
Import Module from Project	Opens a dialog box to select another project. This is done by selecting the `smj` file. This will add a link to the active project for each module in the external project. The files of an imported project are be read only, since the active project is not the owner. An imported module node has a different color than the project nodes.
Build	This launches the *Build* process. The messages generated by the process will be displayed in the output window.
Rebuild All	This launches the *Rebuild All* process. This will rebuild all portions of a project even if normally they would not need to be built. The messages generated by the process will be displayed in the output window.
Execute	This executes the project. If it needs to be built first, then it will be built prior to execution. The messages generated by the process will be displayed in the output window.
Settings	Opens the *Project Settings* dialog box.
Properties	Opens the *Properties* dialog box. In this case, the description of the project file path, the date of last modification of the file, and the path of the byte-code file that is generated as a result of the project build are shown.

The Module Node

There are two types of module nodes, project nodes and imported module nodes. They are shown in different colors.

Table 4.2. Menu Options

Menu Item	Description
Rename Module	Available only for the project modules. This opens a dialog box to change the name of the module and the name of its main source code file.
Remove Module	If the node is an imported module, the link to the module from another project will be removed. If the node is a project node, a dialog box to remove the module will be opened. The dialog box includes an option to remove the module folder and all of its contents.
Create SIMPOL File	Available only for the project modules. This opens a dialog box to create an empty source code file within the module.
Properties	Opens the *Properties* dialog box. In this case, the module folder path and the date of last modification of the module are shown.

The Main Source Code Node

This represents the main source code file. It is the root of the source code files of the module.

Table 4.3. Menu Options

Menu Item	Description
Open File	Opens the file in the editor.
Compile File	The file is compiled.
Execute File	The compiled file associated with the main source code file is executed. If necessary, the source will be compiled first.
Properties	Opens the *Properties* dialog box. There are two tabs. The first one displays the file path, the date of last modification of the file, and the location of the byte-code file after compilation. In the second tab we can see a list with all of the files that are included in the module. If is there a circular path when including files, then the wrong path is shown in this tab. An example of a circular path might be: file A includes file B. File B includes file C and file C includes File A.

Source Code Nodes

This represents a source code file. These are all of the `sma` or `smu` files that are not the main source code file of a module.

Table 4.4. Menu Options

Menu Item	Description
Open File	Opens the file in the editor.
Delete	Shows a dialog box and asks for confirmation to delete the file and remove the reference to it in the project.
Properties	Opens the *Properties* dialog box. This dialog box displays the file path and the date of last modification of the file.
Regenerate Server Page	This is available only for the source code files that are the output file of a server page file compilation. If the source code file has been modified, the regenerate process regenerates the associated server page, so that the output of the process is the server page file. For example: `MySPFile.smu` — regenerates to —> `MySPFile.smz`. This is useful when in a server page there is a large block of SIMPOL code. A server page is essentialy a HTML page with blocks of embed

Menu Item	Description
	SIMPOL code. So the editor applies the HTML color coding rules to the document. If we want to have the advantage of the Help Valet with the SIMPOL language portion of the server page, then we have to work with the SIMPOL source code document generated after the compilation of the server page. We can modify the SIMPOL source code in the source code document and take advantage of the help of the SIMPOL color coding rules and the context-sensitive Help Valet, and then we can regenerate the server page and continue working in the server page on the HTML.

Server Pages Node

This node represents the group of all the server page files in the module. These files are located in the module directory.

Table 4.5. Menu Options

Menu Item	Description
Create New Server Page	Opens a dialog box to add a new server page file to the module.
Reload Server Pages	Load all of the module server pages as child nodes of the Server Page nodes.

Server Page Node

This represents a server page file.

Table 4.6. Menu Options

Menu Item	Description
Open File	Opens the file in the editor.
Compile File	The server page is compiled.
Delete	Shows a dialog box and asks for confirmation to delete the file and remove the reference to it in the project.
HTML Viewer	Opens the HTML viewer and loads the server page into it.
Properties	Opens the *Properties* dialog box. There are two tabs. The first one displays the file path, the date of last modification of the file, and the location of the byte-code file after compilation. In the second tab we can see a list with all of the files that are included in the server page. If is there a circular path when including files, then the wrong path is shown in this tab. An example of a circular path might be: file A includes file B. File B includes file C and file C includes File A.

Other Nodes

This node represents any node related to a file with an extension other than `smj`, `smu`, `sma`, or `smz`. Typically, it is a file with a block of HTML that is included in a server page.

Table 4.7. Menu Options

Menu Item	Description
Open File	Opens the file in the editor.
Delete	Shows a dialog box and asks for confirmation to delete the file and remove the reference to it in the project.

Menu Item	Description
Properties	Opens the *Properties* dialog box. This dialog box displays the file path and the date of last modification of the file.

Type View

This window displays the content of the project library files in a hierarchical or tree layout. A library file or library is the byte-code file generated after compilation of a modules main source code file or after the build of a project. So a library is always a file with the extension `smp` or `sml`. In the type view tree there is a library node for each library in the project. Each library node has a child node for each function and for each type that is in the library. Each function node also has a child node for each argument of the function. The first child node is the first argument, the second child node is the second argument and so on. Each type node has a child node for each property and method. As with the functions, each method has a child node per argument.

Type View Nodes

Library Node

A library node represents a link to a library file.

There are three types of libraries:

• SIMPOL language library

• Project module library

• External linked library

SIMPOL Language Node

This node contains the internal type and function information for the SIMPOL language. For example, function `.toval` or type cgicall. The label for this node is `<smpol>`.

Project Module Nodes

There is one node of this type for each module in the project or imported module. These libraries contain the information about all of the exported and non-exported functions and types. The name of the node is the name of the library file, for example: `MyLibrary.sml`. If the right mouse button is clicked on this type of node then the library file path and date of the last modification will be displayed.

External Module Nodes

This contains the information about all of the exported functions and types from an external library. The external library is linked to the project output file, when the project is built. The name of the node is the name of the library file and is shown between angular brackets. If the right mouse button is clicked on this type of node then the library file path and date of the last modification will be displayed.

Function Node

A `function` node represents a link to a function description. It also represents a library function if it is a `library` child node, or a type method, if it is a `type` child node.

Type Node

A `type` node represents a link to a type description.

Element Node

An element node represents a link to an element description. An element can be a type property, a function or method parameter, or a type tag.

The Output Windows Control Bar

This is the location where the IDE communicates results to the user.

Output Window

This window displays information generated by the application in general. For example, the messages generated by a build, or the results of an executed SIMPOL program are shown in this window.

Debug Window

This window displays information generated by the debugger.

Find in Files Window

This window displays information generated by the `Find in files` process. It displays a line for each match found. In each of those lines is shown the file path and the line where the match was found. If we double-click on a line, the file will be opened in the editor and the line where the match was found will be shown.

The Call Stack Control Bar

This is active only when the debugger is running and the thread with the focus is suspended. This is a read-only window that displays the stack of function calls of the thread that is suspended. The bottom function is always the first function the thread began to execute. If the thread is the main thread ("Thread 1"), this function will be `main`. The top function is always the function where the execution pointer is currently located. Double-clicking on a line in the `Call Stack` window will cause the source code line that is displayed to be executed. That source code belongs to the function selected in the `Call Stack` window.

The Variables Control Bar

This is active only when the debugger is running and the thread with the focus is suspended.

The Locals Window

This window is a table with two columns. The first column is `Name` and the second `Value`. This table shows the name and the current value of the local variables for the function selected in the `Call Stack` window. The function can be changed if we double-click on another function in the `Call Stack` window. By default the local variables that are shown are from the function in which the execution pointer is currently located.

The Me Window

This window is also a table with two columns. The first column is `Name` and the second `Value`. This table shows the name and the current value of the properties of a type if the function currently selected in the `Call Stack` window is a method of a type. The method can be changed if we double-click on another method in the `Call Stack` window. By default the type properties that are shown are from the method of a type in which the execution pointer is currently located.

Menus

These menus are located at the top of the application. Any menu when selected displays an options list. Each option performs a specific task.

File Menu

Table 4.8. File Menu Items

Menu Item	Description
New	Creates a new document. A list with the active language extensions is displayed. This list can be changed in the editor/settings menu entry.
Open	Opens an existing document.
Close	Closes the active document.
New Project	Creates a new project.
Open Project	Opens a project. If there is a project already active, it is closed before the new one is opened.
Close Project	Closes the active project.
Save Project As...	Allows user to save the active document with another name in another location.
Save All	Saves all the open documents.
Print...	Prints the active document.
Print Preview	Displays how the active document would look like if it were printed.
Print Setup...	Opens the *Print* dialog box. The print options can be changed there.
Recent Files	Displays a list of the last files opened.
Recent Projects	Displays a list of the last projects opened.
Exit	Quits the application. Prompts to save any modified documents.

Edit Menu

Table 4.9. Edit Menu Items

Menu Item	Description
Undo	Undoes the last action.
Redo	Redoes the previously undone action.
Cut	Cuts the selection and puts it on the Clipboard.
Copy	Copies the selection and puts it on the Clipboard.
Paste	Inserts contents of the Clipboard.
Comment	This menu option is only available if the active document is a SIMPOL source code document (*sma; or smu). It comments the lines selected in the active document. It prefixes the beginning of each line with a double slash string: "//".
Uncomment	This menu option is only available if the active document is a SIMPOL source code document (*sma; or smu). It uncomments the lines selected in the active document. It removes the double slash comment prefix ("//") from the beginning of each line.
Find	Searches for a string in the active document. The *Find* dialog box contains the following input boxes: 1. *Match whole word only*: If it is checked, the *Find what* entry will have to match a whole word in the document text to be found. 2. *Match case*: If this is checked, the search will be case sensitive.

Menu Item	Description
	3. *Regular Expression*: If it is checked, the *Find what* entry will be treated as an standard regular expression. The direction of the search can be "up" or "down". It can be changed using the appropriate radio buttons. There is also the button "Mark All". If this is pressed, a bookmark will be added to each line that contains the search string.
Find In Files	Searches for a string in multiple files. This is a very powerful search tool that can find the search string in multiple files and folders. The Find in Files dialog contains the following input boxes: 1. *Find what*: Enter the search text here. 2. *In files/file types*: Here you must enter the names of the target files you wish to be searched. The names of the files have to be separated by semicolons. The wildchar "*" can be used. Example: *sma;*smu 3. *In folder*: Enter the name of the search folder here. The *Find in Files* dialog box also contains the following check boxes: 1. *Match case*: If this is checked, the search will be case sensitive. 2. *Regular Expression*: If this is checked, the *Find what* entry will be treated as a standard regular expression. 3. *Look in subfolders*: If this is checked, files in the subfolders of the target folder will also be included in the search. 4. *Look in project*: If this is checked, the *In files/file types* and *In folder* values will be discarded and the search will only take place in the "sma", "smu" and "smz" files that belongs to the project.
Replace...	It opens the *Replace* dialog box. It looks similar to the *Find* dialog box, but it has another entry to introduce the text that should replace the search text.
Insert Code Block	This is only active if the active document is a SIMPOL server page (smz). It inserts a new empty code block in the document before the line with the caret. A code block is a place to write SIMPOL language in a SIMPOL server page; see Example 4.5, "Code block limits".

Example 4.5. Code block limits

```
<%'------------------------- begin code --------------------------
'------------------------- end code  -------------------------%>
```

View Menu

Table 4.10. View Menu Items

Menu Item	Description
Standard Toolbar	Shows the standard toolbar.
Edit Toolbar	Shows the edit toolbar.

Menu Item	Description
Debug Toolbar	Shows the debug toolbar.
Status Bar	Shows the status bar. The status bar is a thin bar at the bottom of the application frame. On the left, the bar displays small pieces of information when the mouse cursor is moved over a toolbar button, menu item, etc. On the right it displays the line and column where the caret is located in the active document.
Full Screen	Expands the document editor to the whole screen.
Projectspace	Adds the "Project Space" control bar to the application frame. Typically, the bar is located at the left of the application frame.
Output	Adds the "Output" control bar to the application frame. Typically, the bar is located at the bottom of the application frame.
Call Stack	Adds the "Call Stack" control bar to the application frame. Typically, the bar is located at the bottom of the application frame.
Variables	Adds the "Variables" control bar to the application frame. Typically, the bar is located at the bottom of the application frame.

Project Menu

Table 4.11. Project Menu Items

Menu Item	Description
Build	Builds the active project. Will only compile files modified since the last build.
Rebuild All	Rebuilds the whole active project. This causes all the files of the project to be compiled.
Execute	Executes the active project. If any files belonging to the project have been modified since the last build, the project will be rebuilt before being executed. The execution result will be displayed in the output window.
Stop building	Stops the current build.
Stop executing	Stops the currently executing program.
Refresh documents	Reloads all the documents opened in the editor.
Settings	Opens the *Project Settings* dialog box.

Debug Menu

Table 4.12. Debug Menu Items

Menu Item	Description
Start debugging	Starts debugging the SIMPOL project. The debugger is launched and the execution is stopped just before the first line in the "main" function code.
Stop debugging	Stops debugging the SIMPOL project.
Continue thread execution	Continues the execution of the thread that is the focus of the debugger.
Break thread execution	Breaks the execution of the thread that is the focus of the debugger. The debugger displays the source code line for the current instruction.
Show Next Statement	Displays the statement that will be executed next.

Menu Item	Description
Step Into	Runs the next statement. If the next statement is a function call, and the source code for the called function is available, the debugger will stop just before the execution of the first statement in the called function.
Step Over	Runs next statement.
Step Out	Runs the program to the end of the current function and steps out to the caller function. Execution will break upon return to the caller.
Run to Cursor	Runs the program to the line containing the cursor.
Insert/Remove Breakpoint	Inserts or removes a breakpoint at the source code line containing the cursor.
Set Next Statement	Changes the execution pointer to another position. The new position is always the beginning of a code line in the function that is being executed.
Thread Manager	Opens the *Thread Manager* dialog box. This option is only available when debugging a program.
Breakpoint Manager	Opens the *Breakpoint Manager* dialog box.
Watch	Opens the *Watch Window* dialog box. This option is only available when debugging a program.
Profile	Opens the *Profile* dialog box. This option is only available when debugging a program.

Document Menu

Table 4.13. Document Menu Items

Menu Item	Description
Compile File	Compiles the active document file if it is a SIMPOL source code file, or a SIMPOL server page file.
Execute File	Executes the byte-code file associated with the active document if it is a SIMPOL source code file. The execution will produce an error if the file doesn't have a "main" function.
Command Line...	Opens a dialog box for the user to add parameters. These parameters will be passed to the "main" function when we execute a SIMPOL file using the *Execute File* option. The parameters are separated by one or more whitespaces. If a parameter contains whitespaces, it should come between a pair of double or single quotes. Example: ``` function main prototype: main(string s, string s2) command line: "hi bye" 123 ```
DOS Newline	This means that the lines in the file are separated by "\r\n". It can be changed to Unix or Mac style.
Unix Newline	This means that the lines in the file are separated by "\n". It can be changed to DOS or Mac style.
Mac Newline	This means that the lines in the file are separated by "\r". It can be changed to DOS or Unix style.
Unicode format	This is option is designed to allow the user to change the ASCII/Unicode format of a file. If it is checked it means that the file content is unicode, if it is not checked then the content is ASCII. Unicode files have a byte or-

Menu Item	Description
	der mark at the beginning of the file and each character is stored in two bytes. ASCII files do not have a byte order mark, and each character is stored in a single byte.
Trim Trailing White Spaces on Lines	Removes all the whitespaces and tabulator characters at the end of each line in the active document.
HTML Viewer	Launches the HTML viewer. This option is only available when the active document is an HTML file or a SIMPOL server page.

Window Menu

Table 4.14. Window Menu Items

Menu Item	Description
New Window	This is only available when there is at least one active document. It creates a duplicate copy of the currently opened window.
Cascade	This is only available when there is at least one active document. It arranges the windows in the editor frame as overlapping tiles.
Tile	This is only available when there is at least one active document. It arranges the windows in the editor frame as non-overlapping tiles.
Arrange Icons	This is only available when there is at least one active document. It arranges icons at the bottom of the window.
Close All	Closes all open documents.

Tools Menu

Table 4.15. Tool Menu Items

Menu Item	Description
Project Report	This is only available if there is an active project open. It opens a dialog box where there are five options to generate documentation from the active project. The options are as follows: *Project summary in TEXT format* Creates a text document briefely describing the active project. It contains the project file paths, the project settings and the project functions and types prototypes. *Project summary in HTML format* Same as the first option but in HTML format. *Project description in XML format* Creates a XML document with all the active project information. *Project library information in HTML format* Creates a HTML document describing the functions and types that belong to the active project. *SIMPOL library information in HTML format*

Menu Item	Description
	Creates a HTML document describing the functions and types that belong to the SIMPOL internal library.

Projects Report	Opens a dialog box to generate documentation for multiple projects. There are two entries in the dialog box. One to enter an input folder. The projects report process will make a report per each project found in this folder or in any subfolders within the input folder. The second entry is the output folder. This is the place where the process will leave all the reports. All the reports are HTML files and an "index.html" file is created with the report list. A report for the SIMPOL internal library is also created. A "logfile.txt" is created with the incidences that happen in the process. Any report contains all the function and type information of each project. The powerful issue is that the process creates a HTML link in every library, function, type, method, property, parameter, tag, etc., so from any report we can go to another report were the description is located. The process also adds the code lines just above a function or type declaration that start with a double slash mark ("//" SIMPOL comment mark).
	
	In the example below, the file "MyFile.sma" contains these two functions:
	
	<pre>// This function returns a string
	// The string is "Hello"
	function f1()
	end "Hello"
	
	// This function returns a string
	// The string is "Bye"
	function f2()
	end "Bye"</pre>
	In the documentation generated for the function f1 we will find this piece of text: 'This function returns a string. The string is "Hello" '.
	
	And in the documentation generated for f2 we will find the equivalent piece of text: 'This function returns a string. The string is "Bye" '.

Call Graph	This is only available if there is an active project. It opens the *Call Analyzer* dialog box. The dialog box shows a tree diagram of the calls among functions in the project. In this case the parent function calls the child function. .
Caller Graph	This is only available if there is an active project. It opens the *Call Analyzer* dialog box. The dialog box shows a tree diagram of the calls among functions in the project. In this case the child function calls the parent function. .
Check Project File	This is only available if there is no active project. It opens the *Check Project File* dialog box.
Open file as binary	Opens a dialog box that allows the user to open a file as binary. This means that a document will be created with the data of the file selected. This document contains two lines for each 32 bytes read in the file. The first line shows the value of each byte in hexadecimal format, and the second shows the ASCII translation of each byte. The address of the byte in the file is shown on the left side of the first line, so it is easy to follow the binary representation.

Menu Item	Description
	The document created will have a `bin` file extension and the appropiate color coding syntax. This makes it very easy to read the document. The searching function becomes quite a powerful tool in the binary document. If the "Find" menu option is pressed, a modified *Find* dialog box opens to search the information in this type of document. This dialog box has a "Find what" entry to enter the search text. Any text entered will be translated into ASCII, and searched for in the document. There is a "Unicode" checkbox which, when checked, will instruct the application to translate the search text into unicode before searching the document, i.e. it will allow two bytes per character rather than one. In such a case, the rightmost byte is zero. There is also a "Match case" checkbox. If this is checked, the search will only return results where the case exactly matches that of the search text. Users can also use regular expressions to find information in the binary file by checking the "Regular expression" checkbox.
Options	Opens the *Application Options* dialog box.

Help Menu

Table 4.16. Help Menu Items

Menu Item	Description
About SIMPOL IDE...	Displays program information, version number and copyright.
Keyboard Map...	Shows the application keyboard map.

Tool Bars

There are three toolbars available in the application. Almost all of the buttons have an associated entry in the menus, so for help regarding the use of a button, please refer to the help for the associated menu command.

Standard Toolbar

This is the standard toolbar, similar to that found in many applications. It contains the following buttons:

• New

• Open

• Save

• Save All

• Cut

• Copy

• Paste

• Print

• About

• Find in Files

- HTML viewer

- Close All

Edit Toolbar

This toolbar contains the following buttons:

- Undo

- Redo

- Find

- Previous Text Found - *Moves to previous text found.*

- Next Text Found - *Moves to next found.*

- Toggle Bookmark - *Toggles a bookmark for the current line on and off.*

- Next Bookmark - *Moves to the line containing the next bookmark.*

- Previous Bookmark - *Moves to the line containing the previous bookmark.*

- Clear All Bookmarks - *Clear all bookmarks in the active window.*

Debug Toolbar

This toolbar contains the following buttons:

- Start Debugging

- Stop Debugging

- Continue Thread Execution

- Break Thread Execution

- Show Next Statement

- Step Into

- Step Over

- Step Out

- Run To Cursor

- Insert/Remove Breakpoint

- Set Next Statement

- Threads

- Call Stack

- Variables

- Watch

Important Dialogs

The following is a description of the main application dialog boxes.

Breakpoint Manager

This dialog box manages the active project breakpoints status. A breakpoint is a mark added to aid debugging. When the application reaches that line, it will interrupt the execution of the program. The line containing the breakpoint will not be executed.

To add or remove a breakpoint in the active document, move the caret to the line and press the "Inserts/ Remove Breakpoint" button. The breakpoint is displayed in the editor as a maroon circle on the left of the line. When a breakpoint is added, it is active for all the threads of the program. By default there is no condition for the break, i.e. the execution will break in all cases. It is possible to add a break condition, using the "Condition to stop execution", field, which is explained below.

The *Breakpoint Manager* dialog box shows a table with information on all the project breakpoints. There is a row for each breakpoint. The following table explains each component of the *Breakpoint Manager* dialog box:

Table 4.17. Breakpoint Manager Dialog Box

Menu Item	Description
'Enabled' column	Shows if the breakpoint is active or not. It can be toggled.
'File path' column	Full file path of the file that contains the breakpoint.
'File line' column	The line in the source code file were the breakpoint is.
'Thread ID' column	Thread that will "see" the breakpoint. Valid data includes: 'all', '1', '2', '3', '4', etc.
'Condition to stop execution' column	Condition that will be evaluated when the execution get the breakpoint. If the condition evaluates to be true, the execution will be interrupted. If the condition evaluate to be false, the execution will continue uninterrupted. This is a powerful feature and is fully explained in the section called "Expression evaluation help"
'Show source code for selected breakpoint' button	When this button is pressed, the editor opens the file and shows the line where the selected breakpoint is located.

Expression evaluation help

Expression data types

- boolean

- integer

- string

Expression operators

The following is a list of the expression operators available in SIMPOL

Table 4.18. Expression operators

Operator	Syntax	Return Type
+	`integer + integer`	`integer`
–	`integer - integer`	`integer`
*	`integer * integer`	`integer`
/	`integer / integer`	`integer`
<	`integer < integer`	`boolean`

Operator	Syntax	Return Type
>	integer > integer	boolean
<=	integer <= integer	boolean
>=	integer >= integer	boolean
==	integer == integer	boolean
!=	integer != integer	boolean
<>	integer <> integer	boolean
+	string + string	string
==	string == string	boolean
!=	string != string	boolean
<>	string <> string	boolean
and	boolean and boolean	boolean
or	boolean or boolean	boolean
()	Used to group sub-operations	none

NB: `!=` and `<>` are equivalent

Constant Values

Data Type	Accepted Values
boolean	`.true` or `.false`
integer	Any integer >= `-2147483647` and <= `2147483647`
string	Any array of characters in between quotes: `"....."` or `'.....'`

Variable Values

- A variable value is a function local variable or a type property value

Breakpoint Condition

- Must be a boolean expression

Watch Window Expression

- Can be any expression or object reference

Built In Functions

The following is a list of the built-in functions in the SIMPOL language:

Syntax	Return Type	Example	Returned Value
#StrAscii (string input)	string	#StrAscii ("Hello")	68 65 6C 6C 6F
#StrUnicode (string input)	string	#StrUnicode ("Hello")	0068 0065 006C 006C 006F
#StrLength (string input)	integer	#StrLength ("He{d}{a}llo")	11
#StrLengthEx (string input)	integer	#StrLengthEx ("He{d}{a}llo")	7

Syntax	Return Type	Example	Returned Value
#Trace (string text)	.false	Example: #Trace ("Hello")	.false (the text goes to the debug window)
#BoolToStr (boolean input)	string	#BoolToStr (.true)	".true"
#IntToStr (integer input)	string	#IntToStr (123)	"123"
#StrUCase (string input)	string	#StrUCase ("Hello")	"HELLO"
#StrLCase (string input)	string	#StrLCase ("Hello")	"hello"
#StrTrim (string input)	string	#StrLCase (" Hello ")	"Hello"
#StrRight (string input, integer length)	string	#StrRight ("Hello", 2)	"lo"
#StrLeft (string input, integer length)	string	#StrLeft ("Hello", 2)	"He"
#StrMid (string input, integer from, integer length)	string	#StrMid ("Hello", 3, 2)	"lo"
–	–	#StrMid ("Hello", 3, -2)	"el"
#StrFindF (string input, integer from, string match)	integer	#StrFindF ("Hello", 0, "el")	1
–	–	#StrFindF ("Hello", 0, "bye")	-1 (not found)
#StrFindB (string input, integer from, string match)	integer	#StrFindB ("Hello", 5, "el")	1
–	–	#StrFindB ("Hello", 5, "bye")	-1 (not found)
#FileTrace (string path, string text)	.false	#FileTrace ("c:\temp\log.txt", "Hello")	.false (The text is appended to the file "c:\temp\log.txt")
#StrHextToInt (string input)	integer	#StrHextToInt ("FFFE")	65534
#IntToStrHext (integer input)	string	#IntToStrHext (65534)	"FFFE"
#StrRepeat (string input, integer repeat)	string	#StrRepeat ("Hello", 3)	"HelloHelloHello"

Syntax	Return Type	Example	Returned Value
`#VarContentToFile` `(string path,` `string variable)`	`.false`	`#VarContent-` `ToFile ("c:` `\temp\log.txt",` `"MyObject.property1")`	`.false (Useful` `for analyzing` `the content of` `large variables)`
`#VarContentLength` `(string variable)`	`integer`	`#VarContentLength` `("MyObject.property1")`	`12000`
`#VarStructToFile` `(string path,` `string variable)`	`.false`	`#VarStruct-` `ToFile ("c:` `\temp\log.txt",` `"MyObject")`	`.false (The ob-` `ject structure and` `content is saved` `in XML format)`
`#GetCurrent-` `Directory()`	`string`	`#GetCurrent-` `Directory()`	`"c:\temp\"`

Breakpoint Expression Examples

Below are some sample code fragments and some example breakpoint expressions. Breakpoint expressions can only be evaluated on lines of executable code.

```
type TA
  bool m1
  integer m2
  string m3
end type

type TB
  string m4
  TA m5
end type

function example()
  bool b
  integer i
  string s
  TB tb

  ...........
  ...........
end function
```

Table 4.19. Breakpoint Expression Examples

Sample	Return Type
`1 < i and s == "Hi"`	`boolean`
`(1 + 3) * 2 + i >= 3`	`boolean`
`b and .false or tb.m5.m1`	`boolean`
`tb.m5.m1`	`boolean`
`s`	`string`
`tb`	`object reference`
`(b or tb.m5.m1) and tb.m4 == s`	`boolean`
`(i * 3 + 2) + 2 - 14 / tb.m5.m2`	`integer`

Call Analyzer

This is only available if there is an active project. This dialog box represents a tree view of all the calls among functions in the project. Each node represents a function. If the dialog box is opened from "Menu/Tools/CallGraph", the parent function calls the child one. If the dialog box is opened frome "Menu/Tools/CallerGraph", the parent function is called by the child one.

The dialog box has a combo box for the user to select the project function that will be the root of the tree. Once this is selected, the tree will be created with the functions that call or get called by the parent node and so on.

Each node displays the name of the function that it represents. At the bottom of the dialog box the file path of the function is displayed and the function prototype. If the user double clicks on a node, the editor will open the source code file where the function is located and show the function definition. This will happen only if the source code file is available.

Check Project File

This is only allowed if there is no active project. Using this dialog box, the user can check if the structure of a project definition file (a `smj` file) is valid or not.

The dialog box has an entry for the project definition file path. After entering the file path, the button "Check consistency" becomes active. This button will check the consistency of the file and will display the result in the box below the button. There are two buttons below the result box. The first button edits the project file definition. The second button loads the project that will be active if the project definition file is OK. If there are problems with the project definition file, the user has to edit it, make the appropiate changes and save the modified file. After that the user must check the consistency again repeat the process until the project description file is valid.

This dialog box will always show up if the user tries to load a project with an invalid project description file.

Application Options

This dialog box stores general application properties. This information is stored in the windows registry.The following properties can be modified:

Property	Description
SIMPOL compiler file path	Shows the SIMPOL compiler file path. It is a `smp` file with the functionality to compile a source code file (`sma` or `smu`) into a byte-code file (`smp` or `sml`). The SIMPOL compiler is a byte-code file that is executed in the SIMPOL virtual machine as any other `smp` or `sml` file. If the application cannot find the compiler file at the path you specify, it will search in the root directory of the application.
Working Directory	Use this text box to enter the path of the folder you wish to contain your Superbase NG projects. This information will be used by the application to make it easier for the user to handle project information.
MultiLanguage	If this checkbox is checked, the user will be able to work with several different languages in the application. This means the user will be able to edit different types of files, and different color-codig syntax rules will be applied in the IDE. If MultiLanguage is not checked, the application will work only with the SIMPOL language.
Optimize Linker Output	This checkbox affects the project link process. If it is not checked, the application will concatinate all the `smp` and `sml` files. This set of files includes the result of compiling all the project modules, plus all the files that are in the list of files to link. Any of the files to link is the result of an external project build, so it is quite common that the files to link share

Property	Description
	part of its content among them. If we check this option, the application will analyze all the files to concatenate and it will remove all the repeated information in the final project byte-code file result of the project build.
Create application Icon File associations	If this button is pressed, a process will be launched to create the application icon file associations. This process makes appropriate changes to the registry, that allow the operating system to associate an icon for each of the following file extensions: `sma`, `smu`, `smz` and `smj`. Hence, if the user double-clicks on any of these files (in windows explorer etc.) the file will be opened with the SIMPOLIDE application. Options to build, rebuild or execute are displayed in a popup menu if the user right-clicks on the icon of a `smj` file.
Languages	In this area, the user can add, remove or change the properties of all the languages the application can handle. The description of each language is stored in a diferent `.ini`" file. By default the application handles the following languages: Binary, C++, C#, HTML, IDL, JScript, SIMPOL, Superbase, Text, Visual Basic, VBScript and XML.

A language can be activated or deactivated with the "active" checkbox. On the left of this area there is a list of languages. On the right, the path to the selected language description file is displayed and the file extensions associated to the language. If there is more than one, they have to be separated by semicolons. There is also a short description of the selected language.To edit the selected language settings, click on the "Edit Language..." button. |
| *Help* | There is a checkbox to activate the help system. The help system is called by pressing "F1" button from any place in the application.

There are two entries in the help area. In the first entry, the user enters the program that will be opened when the help is invoked. The second entry is the file to open with that program. Example: "Exe File: `C:\Program Files\Internet Explorer\iexplore.exe`", "Command Line: `C:\SIMPOL\docs\index.html`". |
| *Autosave* | If this is selected, the application will save anu active project files and folders to the windows temporary folder.In this area, there are two entries. The first is to indicate how many minutes to wait between autosaves. The second entry is to indicate the maximun number of different copies to be stored in the temporay folder.

For example, if the number of minutes is set to 15 and the maximun number of copies is set to 4, after working with "MyProject" for two hours, there will be 4 copies of the project in the windows temporary folder. The names will be: `MyProject_AutoSave_1`, `MyProject_AutoSave_2`, `MyProject_AutoSave_3` and `MyProject_AutoSave_4`, with the most recent being `MyProject_AutoSave_1`. |
| *Save project documents before build...* | If this box is checked and the user is working in a project, the documents that belong to the project will be saved just before a project build, rebuild, execute or debug start, . |
| *SMA source code file default preference* | If this box is checked, the default source code file for new projects becomes the `sma` type. Any new projects opened will now, by default, use only `sma` files. If this box is unchecked, new projects will, by default, only use `smu` files. These settings can be changed once the project is created by using the *Project Settings* dialog box (Project/Project Settings...). |

Property	Description
	This property will not take effect if the user has the ASCII-only application build.

Languages

This dialog box allows the user to change the settings for a language. The language settings are stored in a `.ini` file. For example, the language settings file of the XML language is `XML.ini`.

This dialog box has the following tabs:

Editor

In this area the user can change the basic aspects of a specific language.

We can personalize the following properties:

Property	Description
Tab size	This changes how far the caret jumps when the user presses tab.
Auto indent	If the user hits the return button with auto indent turned on, the caret will be automatically indented to the same position as the text begins on the line above. Until the user types some text, the indentation is temporary, so if the user opens another active window, the caret will be left aligned when he or she returns.
Show whitespace	If this is checked, the tabulators and whitespaces in the document will be displayed with special characters.(a floating decimal place for whitespace and a ">>" character for tabulators).
Virtual whitespace	If this box is unchecked, the user cannot position the caret after the last blank space or character that has been typed in a line. If it is checked, the user can position the caret anywhere in the editing window and begin typing. The application will fill empty spaces in the line with tablators as necessary.
Replace tabs	If this is checked, all tabulators in the document are replaced by equivelant whitespaces. This will have no visual effect on the document unless you have the "Show whitespace" box checked.
Match case	If it is checked, the editor parser will work in a case sensitive mode, if not, the editor parser works in a non-case sensitive mode.
Font face name	Displays the name of the font that is used in the document.
Font size	Displays the size of the font. The "Font Settings" button can be used to change these options.

Parser

The parser is in charge of recognizing the keywords, string patterns, operators, etc throughout the document text. This information is used by the editor to color the text, following the rules of the specific language. These settings are very important, as they effect how the color-coding of the language functions, so the user has to be completely sure before making a change, especially with the SIMPOL language.

The following properties can be edited:

Property	Description
Operators	Characters that are operators in the language. For example: + − *
Delimiters	Characters that are delimiters in the language.

Property	Description
KWStartChars	Special characters that can be the first character of a keyword.
KWMiddleChars	Special characters that can be in the middle of a keyword.
KWEndChars	Special characters that can be at the end of a keyword.

Keywords

Keywords are special reserved words in a language. For example, in SIMPOL language, kewords include: `for`, `if`, `function` and `while`.

On the left there is a list with all the language keywords. Above the list there are buttons to add and remove keywords. On the right side there is a combobox. The combobox list contains the names of the all the different color groups that can be selected. The user can then choose a color group per keyword.

Colors

Each language has a different set of color groups. SIMPOL, for example, has the following groups: `Comment`, `Keyword`, `Number`, `Operator`, `String`, `SystemFunction`, `SystemType`, `Text`, `TextSelection` and `UserType`.

Each color group has a foreground color and a background color. These two colors can be changed using the appropiate buttons on the right side of the color area.

Property	Description
Operators	Characters that are operators in the language. For example: `+ - *`

New Project Options

This dialog box is used to create a new Superbase NG project. It is used to set the properties of the new project.

These are the options:

Property	Description
Project output type	This is the type of file that the build of a project will generate. It can be `smp` or `sml`. Both are byte-code files that will be run in the SIMPOL virtual machine.
Project source code type	This is the type of source code file that will be used in the project. It can be `sma`, which is an ASCII file, or `smu`, which is a Unicode file.
Project location	The folder where the new project will be created.
Project name	The name of the new project.
Wrapper over SIMPOL code file	This means that the project will be created wrapping the SIMPOL source code file selected. A project with one module will be created, and the SIMPOL source code file will be the module's main source code file.
Get properties from project	If this box is checked, the user has the opportunity to select a project. The new project will inherit the project properties of selected project.

Debug Execution Profile

This option is only available if there is an active project and the user is not debugging it. The information recorded from a debug session is displayed here. It is a very powerful feature that shows the developer the bottle-necks of his SIMPOL program and, as a result, he can remove them and improve the program's performance. At the top left of the dialog box there is a checkbox to enable or disable

this feature. If it is enabled, execution in debug mode will go a bit slower in order to record the function calls, time spent in each function, and so on.

There is a box at the top where the user can enter a number of microseconds. This is the maximum amount of time that will be recorded for a statement being executed. The reason behind introducing this cutoff is that a multitasking operating system can pause the execution of a process in the middle of an statement. In this case the time the statement takes to be executed is actually its own time plus the time the microprocessor doing other things. So if we know, for example, that a part of our statement is going to last less than 500 microseconds, we can set this time as a cutoff. This will probably remove all the time the microprocessor is out of our process in the profile report, or at least, the majority of it.

The most important thing in the dialog box is the table, where the recorded information will be displayed after a debug sesion. There is a row per function. The columns of the table are explained below:

Column	Description
Library	The library file name where the function is.
Function	The name of the function. It can be a global function or an object method.
Call count	The number of times the function is called.
Full time (milliseconds)	The time spent in the function for all the calls. It takes into account the time spent executing the function statement, plus any time spent in functions that were called from within it.
Truncate time (milliseconds)	The "Full time" minus the time spent for each statement bigger that the cutoff.
Error time (milliseconds)	This is an indication of the +/- error in the timing of the function. The actual time taken will be somewhere between "Full time" minus "Error time" and "Full time" plus "Error time".
Block time (milliseconds)	The amount of time for which the execution was blocked. This happens, for example, when the SIMPOL program uses sockets, tables in a data base, etc.

Project Settings

This is only available if there is an active project. It displays the project properties, and allows the user to edit them. The project settings are stored always in a file with `smj` extension. The name of this file is the name of the project.

The dialog box shows the following information:

File Folders

This is a list of the folders that the SIMPOL compiler will use to find the included files. There are two types of included path: absolute file paths and relative file paths.

For example, in a lamda source code file, there could be a line like this:

```
include "c:\projects\myproject\includes\MyFile1.sma"
```

Or a line like this:

```
include "MyFile1.sma"
```

The advantage in the second example is that the path in the source code is not made explicit. Note that the path can have the slash or back slash character depending on the operating system.

If the path is absolute, the compiler will use it and nothing more. If the path is relative, for example `MyFile1.sma`, the compiler will firstly use the folder where the file is being compiled to search for the included file. If the included file is not there, it will search for it in each of the "File folders" folders until it is found.

*`sml` Files to link

A list with the `sml` files that will be included in the output project file.

Targets

A target is a copy of the output project file plus a shebang line that is added at the beginning of the file. A target is typically the output of a CGI Project, and the shebang line is the path to the SIMPOL virtual machine program that will execute the byte-code file. The target file is typically called from the web server, and the web server will take the information from the shebang line to execute the file. The targets are created in the project build process.

Targets is a table with a row per target. There are buttons to add, edit and remove a target. The add and edit target buttons will open the target manager dialog box, where the user can add or modify a target. The *Targets* table has the following colums:

Column	Description
Activate	A checkbox to activate or deactivate the target creation in a project build.
Target	The target file path. E.g. `c:\Apache\bin\MyProject.smp`.
Shebang Line	The shebang line. E.g. `#!c:\SIMPOL\smpcgi32.exe{d}{a}`.
Command line	The parameters that will be passed to the "main" function when the project is executed. The parameters are separated by one or more white-spaces. If a parameter contains whitespaces, they should be entered within double quotes or between single quotes. Example: ``` function main prototype: main(string s, string s2) command line: "hi bye" 123 ```
Output file (*`smp`, *`sml`)	The byte-code file that is generated as a result of a project build. If the project has a "main" function, this output file will have a `smp` extension and could be executed alone in the SIMPOL virtual machine. If the project does not have a "main" function, the output file will have `sml` extension. In this case, the byte-code file acts as a library to be linked to by other projects at design-time, or as a library to be loaded dynamically by a running SIMPOL program.
Source code file preference	The default file extension that the application will use for this project when the user creates a new file or opens a file etc.
Make file	If this box is checked, the application will create two make files in the project build process. One is to be used over Windows platforms' (NMAKE facility) and the other is to be used over the Linux platform's (MAKE facility). This make file has the information to make a project build. The file time dependencies are taken into account when making a project build.
CGI Project	This contains a checkbox and an area to enter information. It is provided in the case the user wishes work with a CGI Project. If the box is checked, the project changes from a normal project to a CGI project. A CGI SIMPOL program is pretended to be called from a web server. These programs have a main function but with an special parameter. This parameter is a reference to a CGICall object that will transport all the information that the web server received from a browser call. The CGI SIMPOL program will perform an action depending on the brower request and will output HTML code embedded in SIMPOL strings to the CGICall object. This HTML code will be returned to the browser that made the call through the web server and will be displayed in the customer's browser as

Column	Description
	a HTML web page. So in the end, what the CGI programer needs is a way to build HTML web pages quickly and a way to modify the web page dynamically, depending on the specific browser request.
	Instead of working directly with the SIMPOL code and trying to imagine how the HTML code embedded in SIMPOL strings will turn out, the programmer can create SIMPOL server pages and work in them. A SIMPOL server page is a HTML page with blocks of SIMPOL code embed. The advantage of working with SIMPOL server pages is that they can be displayed in the HTML viewer of the SIMPOL application. The build process will create the SIMPOL source code associated, and it will be compiled as any other SIMPOL source code file.
	If the CGI Project checkbox is checked, a new folder entitled "Server Pages" is created as a child of every module folder in the Project View Tree. All the SIMPOL server pages that belong to the specific module will be displayed in the folder. A SIMPOL server page is a file with `smz` extension.
	In the CGI Project area the user can set the extension of the source code file that will be generated when compiling a server page. It can be `sma` or `smu`. There is an entry to set the output call format - the "CGICall.output" method that will output an string to the standard output. By default the format is: `cgi.output(%s + "{0D}{0A}",`
	An example is shown below. It is assumed here that the CGICall object is named "cgi". The "%s" are placeholders for the HTML embedded as a string. ```Output CGI format:\n cgi.output(%s + "{0D}{0A}", 1)\nLine in a SIMPOL server page file:\n <TH>Hello </TH>\nLine in the source code file associated\n cgi.output("<TH>Hello </TH>" + "{0D}{0A}", 1)```

Target Manager

This dialog box is opened when the "add/edit target" button in the *Project Manager* dialog box is pressed.

On the left there is a list of target folders and on the right there is a list of shebang lines. Below each list is an edit box where the user can modify the information (target folder or shebang line). There are buttons to add the content of the edit box to the appropiate list, remove an entry in the list or add the content of the entry in the list to the edit box. The list entries are stored in the windows registry. There is also a checkbox to activate or deactivate the target. Finally, there is an entry in the target area to enter the target file name. Note: This is usually the name of the project output, but it can be overwritten.

Watch Window

This is only available when a project is being debugged. It is a very powerful feature that allows the user to evaluate expressions and to display the runtime object content.

The dialog box has an expression entry at the top. At the center of the dialog box there is the object viewer, which displays the result of the expression evaluation in a tree view. If the user enters the

name of a function variable that contains an object as an expression, it will be displayed in the object viewer. The object tree root node represents the object in the variable. There will be a child node per object contained in the root object, and each of these nodes will have a child node per object it contains and so on. This allows the user to inspect the object completely.

The object viewer has two sides. The left side is where the tree nodes are located, and the right side is where the value of the object, if applicable, is displayed. Typically the objects with values are the basic types: boolean, integer, string, number and blob. But each object has an internal value that can or cannot be used. For example, the standard object "date" has an integer as internal value to store the date value.

On each node the name of the variable or type property that holds an object, the type of the variable or type property holder and an internal ID of the current object is displayed. If there are two nodes with the same internal ID in a tree, it means that they refer to the same physical object. All the objects have a child type object. This type object transports information about the structure of the parent object. So the value of the type object will be the type the parent object has at runtime. For example, if the type of a variable is a tag type "type(MyType)" then the object held in the variable can or cannot be a "MyType" object. It is displayed in the child type object.

If an expression is not a variable or a type propery, result will be displayed in the object viewer as a tree with just one node and of another color.

The expressions can contain variables and type property names with boolean, integer and string constants. There are many operators that can be used in an expression, so we can evaluate very complex ones at runtime and retrieve interesting pieces of information. (The expression rules are described in the the section called "Expression evaluation help")

There is an edit box at the botton of the dialog box. When a node of the object tree that holds a basic object is selected, the value will be displayed in the box, and the user can modify it. After a value modification, the user has to press the "Set Value" button if he wants to update the object tree with this new value. After editing and changing several values in the object tree, the user must press the "Save new values" button to save all the changes in the physical objects.

When editing a blob value, a new window appears at the bottom of the dialog. This new window will display the ASCII translation of the binary content of the blob. This is quite useful if the blob contains ASCII information, e.g. ASCII text.

Thread Manager

The *Thread Manager* is the place to modify the running status of a thread whilst debugging a SIMPOL program.

The debugger enumerates the threads sequentially as they are created by the program, with the first one created being known as "Thread 1"

The *Thread Manager* displays the running status of all the threads in the program and the functions that they are executing at the time the thread manager is opened. The user can also suspend or resume any thread and change the debugger focus to another thread. This means that Step Into, Step Out, Run to Cursor etc. will affect this new thread.

Keyboard Shortcuts

Below is a list of the keyboard shortcuts available in the IDE. This list can also be viewed by choosing "Keyboard Map..." from the help menu.

Edit Shortcut Keys

Keys	Description
Ctrl+Shift+8	Toggle view whitespace

Keys	Description
Ctrl+A	Select all
Ctrl+C	Copy
Ctrl+F	Opens "Find" Dialog
Ctrl+H	Opens "Replace" Dialog
Ctrl+J	Comment
Ctrl+Shift+J	Uncomment
Ctrl+K	Inserts a code block in a server page before the current line
Ctrl+L	Cut line
Ctrl+Shift+L	Delete Line
Ctrl+U	Make selection lowercase
Ctrl+Shift+U	Make selection uppercase
Ctrl+V	Paste
Ctrl+Shift+W	Select word
Ctrl+X	Cut
Ctrl+Y	Redo
Ctrl+Z	Undo
Back	Delete previous character
Ctrl+Back	Delete word to start
Alt+Back	Undo
Alt+Shift+Back	Redo
Delete	Delete next character
Ctrl+Delete	Delete word to end
Shift+Delete	Cut selection
Down	Line down
Ctrl+Down	Scroll window one line down
Shift+Down	Select up to line end
End	Go to line end
Ctrl+End	Go to document end
Shift+End	Select up to line end
Ctrl+Shift+End	Select up to document end
Escape	Clear selection
F1	Help
F2	Bookmark next
Ctrl+F2	Bookmark toggle
Shift+F2	Bookmark previous
Ctrl+Shift+F2	Bookmark delete all
F3	Find next
Ctrl+F3	Find next word
Shift+F3	Find previous
trl+Shift+F3	Find previous word
Ctrl+F6	Next pane

Keys	Description
Ctrl+Shift+F6	Previous pane
F9	Break point toggle
Home	Beginning of line
Ctrl+Home	Document start
Shift+Home	Select back to line start
Ctrl+Shift+Home	Select back to document start
Insert	Indicator OVR
Ctrl+Insert	Copy
Shift+Insert	Paste
Left	Character left
Ctrl+Left	Word left
Shift+Left	Select character left
Ctrl+Shift+Left	Select back to word start
Next	Page down
Shift+Next	Select page down
Prior	Page up
Shift+Prior	Select page back
Return	New line
Right	Character right
Ctrl+Right	Word right
Shift+Right	Select character right
Ctrl+Shift+Right	Select up to word end
Tab	Insert one tab
Shift+Tab	Move one tab back
Up	Line up
Ctrl+Up	Window scroll one line up
Shift+Up	Select back to line start

File Shortcut Keys

Keys	Description
Ctrl+N	Creates a new file
Ctrl+O	Opens the file open dialog box
Ctrl+P	Opens the print dialog box
Ctrl+S	Saves the current file
F7	Compile
Ctrl+F5	Execute
F8	Command line dialog box

Project Shortcut Keys

Keys	Description
Ctrl+B	Build

Keys	Description
Ctrl+R	Rebuild all
Ctrl+E	Execute

Intellisense Shortcut Keys

Keys	Description
Ctrl+TAB	Shows function argument list
Ctrl+F7	Shows intrinsic type list
Ctrl+Shift+F7	Shows user defined type list
Ctrl+F8	Shows intrinsic function list
Ctrl+Shift+F8	Shows user defined function list

Call Graph Shortcut Keys

Keys	Description
Ctrl+F9	Shows graph of functions that the selected function calls
Ctrl+Shift+F9	Show graphs of functions that call the selected function

Debugger Shortcut Keys

Keys	Description
F4	Starts debugging
Shift+F4	Stops debugging
F5	Continues thread execution
Alt+Num*	Shows next statement
F11	Step into
F10	Step over
Shift+F11	Step out
Ctrl+F10	Run to cursor
F9	Insert/remove breakpoint
Alt+F9	Opens the breakpoint manager
Shift+F9	Watch