PULSONIX

Pulsonix Spice

Scripting Language Reference

Copyright Notice

Copyright © 2001-2014 WestDev Ltd. and SiMetrix Technologies Ltd
Pulsonix is a Trademark of WestDev Ltd. All rights reserved. E&OE

Copyright in the whole and every part of this software and manual belongs to WestDev Ltd. and may not be used, sold, transferred, copied or reproduced in whole or in part in any manner or in any media to any person, without the prior written consent of WestDev Ltd. If you use this manual you do so at your own risk and on the understanding that neither WestDev Ltd. nor associated companies shall be liable for any loss or damage of any kind.

WestDev Ltd. does not warrant that the software package will function properly in every hardware software environment.

Although WestDev Ltd. has tested the software and reviewed the documentation, WestDev Ltd. makes no warranty or representation, either express or implied, with respect to this software or documentation, their quality, performance, merchantability, or fitness for a particular purpose. This software and documentation are licensed 'as is', and you the licensee, by making use thereof, are assuming the entire risk as to their quality and performance.

In no event will WestDev Ltd. be liable for direct, indirect, special, incidental, or consequential damage arising out of the use or inability to use the software or documentation, even if advised of the possibility of such damages.

WestDev Ltd. reserves the right to alter, modify, correct and upgrade our software programs and publications without notice and without incurring liability.

Microsoft, Windows, Windows NT and Intellimouse are either registered trademarks or trademarks of Microsoft Corporation.

All other trademarks are acknowledged to their respective owners.

Pulsonix, a division of WestDev Ltd.

Printed in the UK. Issue date: 24/03/14 iss 3

Pulsonix
20 Miller Court
Severn Drive
Tewkesbury
Tewkesbury Business Park
Glos, GL20 8DN
United Kingdom

Phone +44 (0)1684 296 551
Fax +44 (0)1684 296 515
Email sales@pulsonix.com
Support support@pulsonix.com
Web www.pulsonix.com

Contents

CONTENTS **5**
CHAPTER 1. INTRODUCTION **13**
Overview 13
Support for the Scripting language 13
CHAPTER 2. THE PULSONIX SPICE SCRIPT LANGUAGE **15**
A Tutorial 15
Example 1: Hello World! 15
Example 2: An Introduction to Loops 15
Example 4: Script to curve trace a BJT 17
Variables, Constants and Types 20
Variable names 21
Types 21
Constants 21
Creating and Assigning Variables 22
New line Character 22
Vectors 22
Scope of Variables. Global Variables 23
Empty Values 23
Empty Strings 23
Quotes: Single and Double 24
Expressions 24
Operators 25
Functions 25
Braced Substitutions 26
Bracketed Lists 26
Type conversion 27
Aliases 27
Statements and Commands 27
Commands 27
Command Switches 27
If Statement 28
While Statement 29
For Statement 29
Script Statement 29
Exit Statement 30
Accessing Simulation Data 30
Overview 30
Groups 30
Collections 31
Multi-division Vectors 31
Multi-division Vectors in Functions 31
Vector References 33
Physical Type 33
User Interface to Scripts 33
Dialog Boxes 33
User Control of Execution 34
Errors 34
Syntax Errors 34
Execution Errors 34
Error Messages 34
Executing Scripts 35
Script Arguments 35
Built-in Scripts 36
Debugging Scripts 36
Startup Script 37
CHAPTER 4. FUNCTION REFERENCE **39**

abs ... 39
AddRemoveDialog ... 39
arg ... 40
arg_rad ... 40
Ascii ... 41
atan ... 42
atan_deg ... 42
BoolSelect ... 42
CanOpenFile ... 43
ChangeDir ... 43
Char ... 44
ChooseDir ... 44
CloseEchoFile ... 44
CollectionName ... 44
ComposeDigital ... 45
CopyURL ... 46
cos ... 46
cos_deg ... 47
Date ... 47
dB ... 48
DefineCurveDialog ... 48
DescendDirectories ... 49
diff ... 50
EditAxisDialog ... 50
EditCrosshairDimensionDialog ... 51
EditCurveMarkerDialog ... 52
EditDeviceDialog ... 52
EditFreeTextDialog ... 53
EditGraphTextBoxDialog ... 53
EditLegendBoxDialog ... 53
EditSelect ... 54
EnterTextDialog ... 54
Execute ... 55
ExistDir ... 55
ExistFunction ... 55
ExistVec ... 56
EXP ... 56
FFT ... 56
Field ... 57
FindModel ... 57
FIR ... 57
Floor ... 58
FormatNumber ... 58
Fourier ... 59
FourierOptionsDialog ... 59
FourierWindow ... 60
FullPath ... 60
GenPrintDialog ... 60
GetAllCurves ... 61
GetAllYAxes ... 61
GetAnalysisInfo ... 62
GetAxisCurves ... 62
GetAxisLimits ... 62
GetAxisType ... 63
GetAxisUnits ... 63
GetColours ... 63
GetColourSpec ... 63
GetConfigLoc ... 64
GetConvergenceInfo ... 64
GetCurDir ... 65
GetCurrentGraph ... 65
GetCursorCurve ... 65
GetCurveAxis ... 65

GetCurveName 65
GetCurves 66
GetCurveVector 66
GetDatumCurve 66
GetDeviceDefinition 66
GetDeviceInfo 67
GetDeviceParameterName 67
GetDriveType 68
GetEnvVar 69
GetFile 69
GetFileCd 69
GetFileExtensions 70
GetFileSave 70
GetFonts 70
GetFontSpec 71
GetGraphObjects 71
GetGraphObjPropNames 71
GetGraphObjPropValue 71
GetGraphTitle 73
GetGroupInfo 73
GetGroupStepParameter 73
GetGroupStepVals 73
GetInstanceParamValues 74
GetInternalDeviceName 74
GetLastError 75
GetLegendProperties 75
GetMenuItems 75
GetModelFiles 76
GetModelName 76
GetModelParameterNames 76
GetModelParameterValues 76
GetModelType 77
GetNonDefaultOptions 77
GetNumCurves 77
GetOption 78
GetPath 78
GetPlatformFeatures 78
GetPrinterInfo 78
GetPrintValues 79
GetSelectedCurves 79
GetSelectedGraphAnno 79
GetSelectedYAxis 79
GetSimConfigLoc 80
GetSimulationInfo 80
GetSimulationSeeds 80
GetSimulatorOption 80
GetSimulatorStats 81
GetSimulatorStatus 81
GetSoaResults 82
GetSystemInfo 82
GetUserFile 82
GetVecStepParameter 85
GetVecStepVals 85
GetWindowNames 85
GetXAxis 85
GraphLimits 85
GroupDelay 86
Groups 86
GroupsInCollection 86
HasLogSpacing 86
Histogram 87
Iff 87
IIR 87

im ... 89
imag ... 89
InputGraph ... 89
Integ ... 89
Interp ... 89
IsComplex ... 90
IsFullPath ... 90
IsModelFile ... 90
IsNum ... 91
IsScript ... 91
IsStr ... 91
Length ... 91
ListDirectory ... 92
ln ... 92
Locate ... 92
log ... 92
log10 ... 93
mag ... 93
magnitude ... 93
MakeCollection ... 93
MakeDir ... 93
MakeString ... 94
Max ... 94
Maxidx ... 94
Maxima ... 94
Maximum ... 95
MCOptions ... 95
mean ... 96
Mean1 ... 96
MessageBox ... 96
Mid ... 97
Minidx ... 98
Min ... 98
Minima ... 98
Minimum ... 99
ModelLibsChanged ... 99
norm ... 99
NumDivisions ... 99
NumElems ... 99
OpenEchoFile ... 100
Parse ... 100
ParseParameterString ... 101
PathEqual ... 102
ph ... 102
phase ... 102
phase_rad ... 102
PhysType ... 103
Progress ... 103
QueryData ... 104
RadioSelect ... 105
Range ... 105
re ... 106
ReadClipboard ... 106
ReadConfigSetting ... 106
ReadFile ... 107
ReadIniKey ... 107
ReadRegSetting ... 108
real ... 108
Ref ... 108
RefName ... 108
RelativePath ... 109
RemoveModelFile ... 109
RestartTranDialog ... 109

Rms ... 109
RMS1 ... 110
rnd ... 110
RootSumOfSquares ... 110
Scan ... 110
ScriptName ... 112
Search ... 112
SearchModels ... 112
SelectColourDialog ... 112
SelectColumns ... 113
SelectDialog ... 113
SelectFontDialog ... 114
SelectRows ... 114
SelGraph ... 115
Shell ... 115
ShellExecute ... 116
sign ... 117
SimulationHasErrors ... 117
sin ... 117
sin_deg ... 117
Sleep ... 117
Sort ... 118
SortIdx ... 118
SplitPath ... 118
sqrt ... 119
Str ... 119
StringLength ... 119
StrStr ... 119
SumNoise ... 120
tan ... 120
tan_deg ... 120
Time ... 120
TranslateLogicalPath ... 121
TreeListDialog ... 121
TRUE ... 122
Truncate ... 122
Units ... 123
unitvec ... 123
UpDownDialog ... 124
Val ... 124
ValueDialog ... 124
vector ... 125
VectorsInGroup ... 125
WriteConfigSetting ... 125
WriteIniKey ... 126
WriteRawData ... 127
WriteRegSetting ... 127
XCursor ... 128
XDatum ... 128
XFromY ... 128
XY ... 128
YCursor ... 128
YDatum ... 129
YFromX ... 129

CHAPTER 3. COMMAND REFERENCE ... 131

Notation ... 131
Abort ... 131
About ... 132
AddCurveMarker ... 132
AddFreeText ... 132
AddGraphDimension ... 133
AddLegend ... 133

AddLegendProp 134
AddTextBox 134
Arguments 135
Cd 135
ChooseColour 135
ClearMessageWindow 135
Close 135
CloseGraphSheet 135
ClosePrinter 135
CollectGarbage 136
CreateFont 136
CreateGroup 136
CreateToolButton 137
CursorMode 137
Curve 138
CurveEditCopy 140
DefButton 140
DefItem 140
DefineToolBar 142
DefKey 144
DefMenu 146
Del 148
DelCrv 148
DeleteAxis 148
DeleteGraphAnno 148
DelGroup 148
DelLegendProp 149
DelMenu 149
Discard 149
Display 149
Echo 150
EditColour 150
EditFile 150
EditFont 150
ExecuteMenu 150
Execute 150
Focus 151
FocusShell 151
Font 151
GraphZoomMode 151
Help 151
HideCurve 152
HighlightCurve 152
Hint 152
KeepGroup 152
Let 152
Listing 154
ListModels 154
ListStdButtonDefs 154
ListStdKeys 154
ListStdMenu 154
LoadModelIndex 154
MakeAlias 155
MakeCatalog 155
MakeCollection 155
MakeTree 155
Mcd 155
Md 156
MessageBox 156
MoveCurve 156
MoveFile 156
NewAxis 156
NewGraphWindow 156

NewGrid ... 156
NewPrinterPage ... 156
NoPaint ... 156
OpenGroup ... 157
OpenPrinter ... 157
OpenRawFile ... 157
OptionsDialog ... 158
Pause ... 158
PlaceCursor ... 158
Plot ... 158
PrintGraph ... 160
Quit ... 160
Rd ... 160
ReadLogicCompatibility ... 161
Redirect ... 162
RegisterDevice ... 162
RegisterUserFunction ... 162
RenameLibs ... 162
RepeatLastMenu ... 163
Reset ... 163
RestartTran ... 163
RestDesk ... 163
Resume ... 163
Run ... 163
SaveDesk ... 165
SaveGraph ... 165
SaveGroup ... 166
SaveRhs ... 166
ScriptAbort ... 166
ScriptPause ... 167
ScriptResume ... 167
ScriptStep ... 167
SelectCursorMode ... 167
SelectCurve ... 168
SelectGraph ... 168
SelectLegends ... 168
Set ... 168
SetCurveName ... 168
SetGraphAnnoProperty ... 169
SetGroup ... 169
SetRef ... 169
SetToolBarVisibility ... 169
SetUnits ... 169
Shell ... 170
ShellOld ... 170
Show ... 170
ShowCurve ... 171
ShowSimulatorWindow ... 171
SizeGraph ... 171
Stats ... 172
Title ... 172
Trace ... 172
UndoGraphZoom ... 172
UnHighlightCurves ... 172
Unlet ... 173
Unset ... 173
ViewFile ... 173
Wait ... 173
Where ... 173
WriteImportedModels ... 173

CHAPTER 4. APPLICATIONS ... 175

User Interface ... 175

User Defined Key and Menu Definitions 175
Rearranging or Renaming the Standard Menus 175
Menu Shortcuts 175
Modifying Internal Scripts 175
Custom Curve/Performance/Histogram Analysis 176
Adding New Functions 176
"measure", "measure_span", "performance" and "mc_histo" Scripts 176
An Example: The "Mean" Function 176
Automating Simulations 177
Overview 177
Running the Simulator 177
Changing Component Values or Test Conditions 177
An Advanced Example - Reading Values from a File 178
Data Import and Export 183
Importing Data 183
Exporting Data 183
Launching Other Applications 183
Data Files Text Format 183
Overview 184
Object Types 185
Properties 185
Graph Object Identifiers - the “ID” 185
Symbolic Values 186
Objects and Their Properties 186
on_graph_anno_doubleclick 192
on_accept_file_drop 192
Overview 193
Defining the Function 193
Registering the Script 193
Example 193
Non-interactive and Customised Printing 193
Overview 193
Procedure 194
Example 194
Modifying Existing Toolbars and Buttons 195
Redefining Button Commands 195
Defining New Buttons and Editing Buttons 195
Creating New Toolbars 196
Pre-defined Buttons 196
INDEX 197

Chapter 1. Introduction

Overview

Pulsonix Spice features a simple interpreted script language, loosely based on BASIC, in which most of the user interface is written.

This manual describes how users sympathetic to the concept of computer programming can develop their own scripts or adapt the user interface by modifying the internal scripts (provided on the Pilsonix CD).

We have identified three main applications for script development although there may be others we haven't thought of. These are:

1. User interface modification perhaps to suit individual taste or for specialised applications.
2. Automated simulations. For example, you may have a large circuit which for which you need to run a number of tests. The simulations take along time so you would like to run them overnight or over a weekend. A simple script can perform this task.
3. Specialised analysis. The curve analysis functions supplied with Pulsonix Spice are all implemented using scripts. You can write your own to implement specialised functionality.

The scripting language is supported by about 150 functions and 100 commands that provide the interface to the Pulsonix Spice core as well as some general purpose functionality.

Support for the Scripting language

The scripting language is only supported via email. It can only be supported in terms of options within it and not on a code basis. If you require additional help with the coding of scripts, you must approach us on a consultancy arrangement.

Chapter 2. The Pulsonix Spice Script Language

A Tutorial

Example 1: Hello World!

Any one who has learnt the 'C' programming language will be familiar with the now celebrated "Hello World" program - possibly the simplest program that can be written. Here we will write and execute a Pulsonix Spice "Hello World" script.

The script is simple:

```
echo Hello World!
```

To execute and run this script start by selecting the **File** menu and **Scripts, New Script** this simply launches notepad with the script directory as its working directory. Type :

```
echo Hello World!
```

Now save the text to a file called HELLO.TXT.

To execute the script, type "hello" at the command line. You should see the message:

```
Hello World!
```

Appear in the message window. Scripts are executed by typing their filename at the command line. If the file has the extension ".txt" the extension can be omitted. You can also assign a key or menu to execute a script. Type at the command line:

```
DefKey F6 HELLO
```

Now press the <F6> function key. The message should appear again. For information on defining menus see User Defined Key and Menu Definitions.

Example 2: An Introduction to Loops

This example adds up all the elements in a vector (or array). To create a vector we will run a simulation on one of the example circuits. The whole process will be put into a script except opening the schematic which we will do manually. (But this can be done from a script as well).

To start with, using Pulsonix open the example circuit \GENERAL\AMP.SCH. Make sure it is selected to run a transient analysis and run the simulation.

Now select **File|Scripts|New Script.** This will open NOTEPAD or your selected text editor with the current directory set to the SCRIPT. Type in the following:

```
let sum = 0
for idx=0 to length(vout)-1
    let sum = sum + vout[idx]
next idx
echo The sum of all values in vout is {sum}
```

Save the script to the file name SUM.TXT. Now type SUM at the command line. A simulation will run and the message:

```
The sum of all values in vout is -1.93804
```

Should appear in the message window. The exact value given may be different if you have modified the circuit or set up different model libraries.

This script introduces four new concepts:

- For loops
- Braced substitutions ({sum} in the last line)
- Vectors (or arrays)
- Accessing simulation data

Let's go through this script line by line.

The line

```
let sum = 0
```

creates and initialises the variable `sum` which will ultimately hold the final result. The next three lines is a simple *for statement*. The variable `idx` is incremented by one each time around the loop starting at zero and ending at `length(vout)-1`. `vout` is a variable - actually a vector - which was generated by the simulator and holds the simulated values of the voltage on the VOUT net. This net is marked with a terminal symbol. `length(vout)` returns the number of elements in `vout` . (1 is subtracted because `idx` starts at 0). In the line:

```
let sum = sum + vout[idx]
```

`vout[idx]` is an indexed expression which returns element number `idx` of the vector `vout`. `sum` is of course the accumulative total. The final line:

```
echo The sum of all values in vout is {sum}
```

contains the *braced substitution* `{sum}`. `sum` is evaluated and the result replaces expression and the braces.

Example 3: Plot the sum of selected curves

This example demonstrates the use of the **GetCurveVector** function. This is a key function used for analysing and processing displayed graphs.

```
** Script to plot sum of selected graph curves
Let selCurves = GetSelectedCurves()
Let numCurves = Length(selCurves)
if numCurves=0 then
    Echo "No selected graph curves available"
    exit script
endif
Let sum=0
for idx=0 to numCurves-1
    Let sum = sum + GetCurveVector(selCurves[idx])
next idx
Curve sum
```

This script introduces *if statements ,arrays, comments* and *functions*.

The first line:

```
** Script to plot sum of selected graph curves
```

is a comment. Any line beginning with a '*' will be ignored.

The next line:

```
Let selCurves = GetSelectedCurves()
```

sets up an array of IDs for all selected curves on currently active graph sheet.

The next line:

```
Let numCurves = Length(selCurves)
```

uses the `Length` function to return the number of elements in the `selCurves` array.

The next line:

```
if numCurves=0 then
```

is the start of an *if statement*. It checks the `numCurves` variable previously set up to see if any curves where selected.

If there were no curves selected, the lines:

```
    Echo "No selected graph curves available"
    exit all
```

will be executed. The first line calls the `echo` command to send to the message window all subsequent text on the same line. The second line is an *exit statement*. In this case it causes execution to abort and the rest of the script will be ignored.

The next line:

```
endif
```

terminates the if statement. For every **`if`** there must be a matching **`endif`** or **`end if`**.

If there were selected curves, the remainder of the script will now be executed. The next line:

```
Let sum=0
```

Creates and initialises the variable `sum` which will ultimately hold the final graph data.

The next line:

```
for idx=0 to numCurves-1
```

starts a for loop. The block of statements between this line and the matching **`next`** will be repeated with the values of `idx` incrementing by 1 each time around the loop until `idx` reaches `numCurves-1`, so the loop is repeated for all elements in `selCurves`.

The next line:

```
    Let sum = sum + GetCurveVector(selCurves[idx])
```

Uses a function **GetCurveVector** to return the data for a curve. `selCurves[idx]` is an indexed expression which returns the element number `idx` of the vector `selCurves`, which is a curve id.

If it's an AC analysis, this data will be complex. (Although the graphs only display the magnitude the data is always stored in its original form which for an AC analysis is complex). So if you add them up their phase will be taken into account. Its possible to test for complex data using the IsComplex() function and then applying mag() if it is. This will then just add up the magnitudes which may or may not be what is wanted depending on the application.

The next line:

```
next idx
```

terminates the **`for`** loop.

The final line:

```
Curve sum
```

Uses the `Curve` command to plot the result. You could do "`Curve sum/numCurves`" to get the average instead. The command line "`Plot sum`" would do the same but create a new graph.

Example 4: Script to curve trace a BJT

This example is included in the scripts directory on the CD, called ‘bjt_ic_vce_sa.sxscr’. It runs a simulation on a dynamically created netlist to plot the I-V characteristics of an NPN or PNP transistor. (A process sometimes known as "curve tracing").

The user will be prompted to enter the device part number and polarity i.e. NPN or PNP, followed by the curve tracing parameters.

```
** Script to curve trace BJT
** Get user's device selection
Let result = EditSelect(['', 'NPN'], ['Part number', 'Polarity
(NPN/PNP)'], 'Define device')
** Exit if no device entered or cancel pressed
if result[0]='' then
```

```
    exit script
endif
Let device=result[0]
** Find polarity
if result[1]='NPN' then
    Let pol=1
elseif result[1]='PNP' then
    Let pol=-1
else
    Echo "Invalid polarity. Must be NPN or PNP"
    exit script
endif
** Find device and try and determine if its a model or
subcircuit.
Let loc = FindModel(device, 'q')
if length(loc)= 0 then
    Let loc = FindModel(device, 'x')
    if length(loc)<>0 then
        Let type='x'
    else
        Echo "Cannot find device" {device}
        exit script
    endif
else
    Let type = 'q'
endif
** Get curve trace parameters from user.
Let options = ValueDialog([1m, 15, 10], ['Max Base Current', 'Max
Collector Voltage', 'Number Steps'], 'BJT Trace Settings')
if Length(options)=0 then
    ** User cancelled
    Exit script
endif
Let IbMax = options[0]*pol
Let VceMax = options[1]*pol
Let numSteps = options[2]
** Create netlist to simulate
Echo /file design.net "* BJT curve trace"
if type='q' then
    Echo /append design.net "Q1 C B 0 0" {device}
else
    ** Subcircuits are assumed to be three terminal.
    ** Script will fail if they aren't
    Echo /append design.net "X1 C B 0" {device}
endif
```

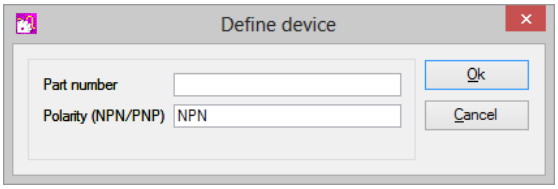
```
Echo /append design.net "Vce C 0 0"
Echo /append design.net "Ib 0 B {Ib}"
Echo /append design.net ".dc Vce 0" {VceMax} {VceMax/50}
Echo /append design.net ".graph Vce#n curveLabel={curveLabel}
graphName={traceid}" ylabel= { '"Collector current ' & device &
'"'}
if not ExistVec('global:traceid') then
    Let global:traceid=1
endif
for global:Ib=IbMax/numSteps To IbMax step IbMax/numSteps
    Let global:curveLabel = 'Ib=' & global:Ib
    Run /file design.net
next global:ib
Let global:traceid=global:traceid+1
```

We will not go through every line of this script in detail, but instead will look at some of the interesting points.

First consider the line:

```
Let result = EditSelect(['', 'NPN'], ['Part number', 'Polarity
(NPN/PNP)'], 'Define device')
```

This uses the **EditSelect** function to get strings from the user. It will show the following dialog:



Next consider the line:

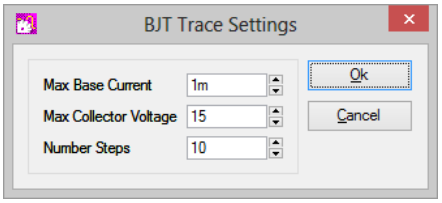
```
Let loc = FindModel(device, 'q')
```

The function **FindModel** returns the file name and line number where the device model is stored. We don't need to know this but we do need to know if its implemented as a subcircuit or as a model. **FindModel** will return empty (Length()=0) if wrong type of device is entered as second argument, so we find whether its a 'q' device or an 'x' device by trying both of them. If both fail then either the device isn't in the library or its not a BJT.

Next look at the line:

```
Let options = ValueDialog([1m, 15, 10], ['Max Base Current', 'Max
Collector Voltage', 'Number Steps'], 'BJT Trace Settings')
```

This uses function **ValueDialog** to get the curve trace parameters from user. This is initialised in the function call with values suitable for a small signal non-darlington device. The following dialog is displayed:



Next look at the line:

```
Echo /file design.net "* BJT curve trace"
```

This uses the `Echo` command to create a netlist file to simulate. '`Echo /file`' and '`Echo /append`' are used in this script to create a file and then append strings to it, although it is

probably more efficient to create a string array with `MakeString` then use '`show /plain /file`' to write the strings to a file.

Now look at line:

```
Echo /append design.net ".graph Vce#n curveLabel={curveLabel}
graphName={traceid}" ylabel= { '"Collector current ' & device &
'"'}
```

The global variable '`global:traceid`' is incremented each time this script is called. This is used as the graphName parameter for .GRAPH and ensures a new graph sheet is created for each run.

The line:

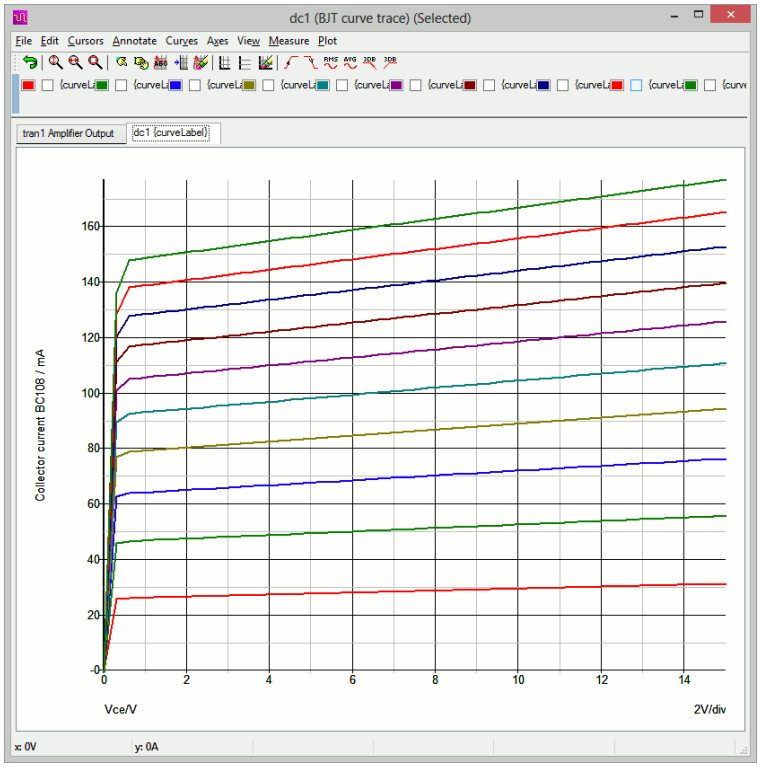
```
for global:Ib=IbMax/numSteps To IbMax step IbMax/numSteps
```

is interesting as it introduces the *for loop* concept of using *step* to increment the variable `global:Ib` by more than one each time.

And lastly, the lines:

```
Let global:curveLabel = 'Ib=' & global:Ib
Run /file design.net
```

Plot each curve as shown in the following graph window:



Variables, Constants and Types

Pulsonix Spice scripts, like all computer programs, process data stored in variables. Variables may hold real, complex or string data and may be scalar - possessing only a single value - or single dimension arrays called vectors[1].

[1] In Pulsonix Spice release 2.0 documentation, variables were always referred to as vectors even if they were actually scalar. This is just different nomenclature. Variables, vectors, scalars and arrays are different names for the same thing.

Variable names

Variables names must be a sequence of characters but the first must be non-numeric. Any character may be used except: \ " & + - * / ^ < > ' @ { } ( ) [ ] ! % ; : |= and spaces.

Although it is legal the following names should be avoided as they are statement keywords:

all

do

else

elseif

end

endif

endwhile

exit

for

if

loop

next

script

step

then

to

while

Types

Variables may have real, complex or string type. Real and complex are self-explanatory. Strings are a sequence of ASCII characters of any length.

Pulsonix Spice does not have an integer type. Although all numbers are represented internally as floating point values, the format used permits integers to be represented exactly up to values of about 2^{52}.

Constants

These can be real complex or string. Real numbers are represented in the usual way but may also contain the engineering suffixes:

a	10^{-18}
f	10^{-15}
p	10^{-12}
n	10^{-9}
u	10^{-6}
m	10^{-3}
k	10^{+3}
Meg	10^{+6}
G	10^{+9}
T	10^{+12}

Complex numbers are represented in the form:

(real, imaginary)

Strings are a sequence of text characters enclosed in single quotation marks. Single quotation marks themselves are represented by two in succession.

Examples

Real:

```
2.3
4.6899
45
1e-3
1.2u
```

Complex

```
(1,1)          means 1+i
(2.34,10)      means 2.34+10i
```

String

```
'this is a string'
'This is a ''string'' '
```

Creating and Assigning Variables

Variables are created and assigned using the Let command. For example:

```
Let x=3
```

assigns the value 3 to the variable x. Note that "Let" is not optional as it is in most forms of BASIC.

You can also assign complex numbers and strings e.g.

```
Let x=(5,1)
Let s='This is a string'
```

All of the above are *scalar* that is they contain only one value. Variables may also be single dimension arrays called *vectors.* Vectors are described in the next section.

New line Character

To enter a new line character use '\\'. If you need a literal double backslash enclose it in quotation marks i.e. "\\". Note however that the use of '\\' doesn't work inside braced substitutions. To use a line feed in a braced substitution, assign the whole string to a variable then put the variable inside the braces. E.g

```
Let error = 'Error:\\Too many nodes'
MessageBox {error}
```

Vectors

Vectors can be created using a *bracketed list*, with a function that returns a vector or by the simulator which creates a number of vectors to represent node voltages and device currents. A bracketed list is of the form:

```
[ expression1, expression2, ...]
```

E.g.

```
let v = [1, 3, 9]
```

Functions and simulator vectors are described in following sections.

Vectors, like other variables may also contain strings or complex numbers but all the elements must be the same *type.*

Individual elements of vectors may be accessed using square brackets: '[' and ']'. E.g.

```
let v = [1, 3, 9]
let a = v[2]
```

'a' is assigned 9 in the above example. Index values start at 0 so the first element (1) is v[0].

It is also possible to assign values to individual elements e.g.

```
let v[2] = 5
```

In which case the value assigned must have the same *type* (i.e. real, complex or string) as the other elements in the vector.

Vectors, like other variables may also contain strings or complex numbers but all the elements must be the same *type.*

Scope of Variables. Global Variables

Variables created using the Let command are only available within the script where the Let command was executed. The variable is destroyed when the script is completed and it is not accessible to scripts that the script calls. If, however, the Let command was called from the command line, the variable is then *global* and is available to all scripts until it is explicitly deleted with the UnLet command.

If a global variable needs to be created within a script, the variable name must be preceded by "global:". For example:

```
Let global:result = 10
```

"global:result" will be accessible by all scripts and from the command line. Further it will be permanently available until explicitly deleted with UnLet. After the variable has been created with the "global:" prefix, it can subsequently be omitted. For example in:

```
Let global:result = 10
Show result
Let result = 11
Show result
```

will display

```
result=10
result=11
```

in the message window. The variable `result` will be available to other scripts whereas if the "global:" prefix had been left off, it would not. Although it is not necessary to include the "global:" prefix except when first creating the variable, it is nevertheless good practice to do so to aid readability of the script.

Empty Values

Some functions return "empty" values when they are unable to produce a return value. An empty value contains no data. An empty value can be tested with the **Length** function which will return 0. All other functions and operators will yield an error if presented with an empty value.

Empty Strings

An empty string is one that has no characters. An empty string can be entered on a command line with the character sequence:

```
{''}
```

Empty strings are not the same as empty values. An empty value has no data at all and will result in an error if supplied to any function other than the Length function.

Quotes: Single and Double

Single quotation marks (') and double quotation marks (") both have a special, but different, meaning in Pulsonix Spice and in the past this has been the source of much confusion. Here we explain what each means and when they should be used.

Single quotes are used to signify a text string in an expression. Expressions are used as arguments to the **Plot**, **Curve**, **Let** and **Show** commands, they are used in braced substitutions and also as the tests for if, for and while statements. These are the only places where you will find or need single quotes.

Double quotes are used in commands to bind together words separated by spaces or semi-colons so that they are treated as one. Normally spaces and semi-colons have a special meaning in a command. Spaces are used to separate arguments of the command while semi-colons terminate the command and start a new one. If enclosed within double quotes, these special meanings are disabled and the text within the quotes is treated as a single argument to the command. Double quotes are often used to enclose strings that contain spaces (see example) but this doesn't necessarily have to be the case.

Examples

```
Let PULSE_SPEC = 'Pulse 0 5 0 10n 10n 1u 2.5u'
```

In the above line we are assigning the variable PULSE_SPEC with a string. This is an expression so the string is in single quotes. **Let** is a command but it is one of the four commands that take an expression as its argument.

```
Plot /name "amp out" vout
```

Plot is a command that creates a new graph for the supplied vector, in this example the vector is the signal name ‘vout’. This example uses a switch to name the curve “amp out”, but this has spaces in so we must enclose it in double quotation marks so that the command treats it as a single string. If there were no quotes, ‘amp’ and ‘out’ would be treated as additional vectors to be plotted rather than being the name of the curve. If an argument contains no spaces or semi-colons then no quotes are necessary although they will do no harm if present.

Where you need both single and double quotes

There are situations where both single and double quotes are needed together. In some of the internal scripts you will find the **Scan** command (page 110) used to split a number of text strings separated by semi-colons. The second argument to **Scan** is a string and must be enclosed in single quotation marks. But this argument is also a semi-colon which, despite being enclosed in single quotes, will still be recognised by the command line interpreter as an end-of-command character. So this must be enclosed in double quotes. The whole expression can be enclosed in double quotes in this case.

If you need a literal quote

If you need a string that contains a double or single quote character, use two of them together.

Expressions

An expression is a sequence of variable *names*, *constants*, *operators* and *functions* that can be evaluated to yield a result. Expressions are required by four commands: Let, Curve, Plot and Show and they are also used in *braced substitutions* and *if statements*, *while statements* and *for statements*. This section describes expression syntax and how they are evaluated.

Examples

```
x+2
(v1_p-vout)*r1#p
idx<15
vout[23]-vout[22]
'Hello ' & 'World'
```

Operators

Loosely, expressions are constants, variables or/and function calls separated by operators. Available operators are:

Arithmetic:

```
+ - * / ^ %
```

'%' performs a remainder function

Relational

```
< >  == <= >=
```

Important: a single '=' can be used as equality operator if used in an *if* or *while* statement. In other places it is an assignment operator and '==' must be used for equality.

Logical

```
AND, OR, NOT,
```

```
&& || !
```

Note: `AND`, `OR`, `NOT` are equivalent to `&& || !` respectively.

String

& (concatenation)

Operator precedence

When calculating an expression like 3+4*5, the 4 is multiplied by 5 first then added to 3. The multiplication operator - '*' - is said to have higher precedence then the addition operator - '+'. The following lists all the operators in order of precedence.

( ) [ ]

Unary - +[2] NOT !

^

* / %

+ -

< > <= >= ==

AND &&

OR ||

&

=

,

Notes

1. A single '=' is interpreted as '==' meaning equality when used in *if statements* and *while statements* and has the same precedence.
2. Parentheses have the highest precedence and are used in their traditional role to change order of evaluation. So (3+4)*5 is 35 whereas 3+4*5 is 23.
3. The comma ',' is used as a separator and so has the lowest precedence.

Functions

Functions are central to Pulsonix Spice scripts. All functions return a value and take zero or more *arguments*. The sqrt function for example takes a single argument and returns its square root. So:

---

[2] E.g. In 5*-3, the '-' is a unary operator - applying to a single value not operating on two values. In this instance '-' has higher precedence than '*'.

```
Let x = sqrt(16)
```

will assign 4 to x.

Functions are of the form

function_name([*argument*, ...])

Examples

Function taking no arguments:

```
NetName()
```

function taking two arguments:

```
FFT( vout, 'Hanning')
```

Functions don't just perform mathematical operations like square root. There are functions for string processing, functions which return information about the simulation data and user interface functions. Complete documentation on all available functions is given in Chapter 3.

Braced Substitutions

A braced substitution is an expression enclosed in curly braces '{' and '}'. When the script interpreter encounters a braced substitution, it evaluates the expression and substitutes the expression and the braces with the result of the evaluation - as if it had been typed in by the user. Braced substitutions are important because, with the exception of the Let, Show, Plot and Curve commands, commands cannot accept expressions as arguments. For example, the Echo command displays in the message window the text following the Echo. If the command "Echo x+2" was executed, the message "x+2" would be displayed not the result of evaluating "x+2". If instead the command was "Echo { x+2 }" the result of x+2 would be displayed.

If the expression inside the braces evaluates to a vector each element of the vector will be substituted. Note that the line length for commands is limited (although the limit is large - in excess of 2000 characters) so substituting vectors should be avoided unless it is known that the vector does not have many elements.

Braced substitutions may not be used in the control expression for conditional statements, while loops and for loops. For example, the following is not permitted

```
if {GetNumCurves()} < 10 then
```

To achieve the same result the result of the braced expression must be assigned to a variable e.g.:

```
let n = {GetNumCurves()}
if n < 10 then
```

Braced substitutions are also permitted in netlists. See the Pulsonix Spice User's manual.

Bracketed Lists

These are of the form

[*expression1*, *expression2*, ...]

The result of a bracketed list is a vector of length equal to the number of expressions separated by commas. There must be at least one expression in a bracketed list - an empty list is not permitted. For example:

```
Let v = [3, 5, 7]
```

assigns a vector of length 3 to v. So v[0]=3, v[1]=5 and v[2]=7. The expressions in a bracketed list may be any type, as long they are all the same. The following for example, is illegal:

```
Let v = [3, 'Hello', 'World']
```

The second element is of type string whereas the first is real. The following example is however legal:

```
Let v = ['3', 'Hello', 'World']
```

3 which is real has been replaced by '3' which is a string.

Type conversion

Most functions and operators expect their arguments to be of a particular type. For example the '+' operator expects each side to be a numeric (real or complex) type and not a string. Conversely, the '&' operator which concatenates strings naturally expects a string on each side. The majority of functions also expect a particular type as arguments, although there are some that can accept any type.

In the event that the type presented is wrong, Pulsonix Spice will attempt to convert the value presented to the correct type. To convert a numeric value to a string is straightforward, the value is simply represented in ASCII form to a reasonable precision. When a string is presented but a numeric value is required, the string is treated as if it were an expression and is evaluated. If the evaluation is successful and resolves to the correct type the result is used as the argument to the operator or function. If the evaluation fails for any reason an error message will be displayed.

Aliases

An *alias* is a special type of string. Alias strings hold an expression which is always evaluated when used. The simulator outputs some of its data in alias form to save memory and simulation time. For example, the currents into subcircuit pins are calculated by adding the currents of all devices within the subcircuit connected to that pin. This current is not calculated during simulation but the expression to perform that calculation is stored as an alias so that it can be calculated if needed.

Statements and Commands

Scripts are composed of a sequence of *statements*. Statements usually comprise at least one *command* and optionally control words such as `if` and `then`. A *command* is a single line of text starting with one of the 100 or so command names listed in the Command Reference.

There are six types of statement. These are:

command statement

if statement

while statement

for statement

jump statement

script statement

Commands

Commands begin with one of the names of commands listed in chapter 4. A command performs an action such as running a simulation or plotting a result. E.g.:

```
Plot v1_p
```

is a command that will create a graph of the vector `v1_p`. The syntax varies for each command. Full details are given in the Command Reference.

All commands must start on a new line or after a semi-colon. They must also end with a new line or semi-colon.

A command statement is a sequence of one or more commands.

Command Switches

Many commands have *switches*. These are always preceded by a '/' and their meaning is specific to the command. There are however four global switches which can be applied to any command. These must always be placed immediately after the command. Global switches are as follows:

- /e Forces command text to copied to command history
- /ne Inhibits command text copying to command history

- /quiet Inhibits error messages for that command. This only stops error message being displayed. A script will still be aborted if an error occurs but no message will be output
- /noerr Stops scripts being aborted if there is an error. The error message will still be displayed.

If Statement

An *if statement* is of the form:

if *expression* **then**

statement

endif

or:

if *expression* **then**

statement

else

statement

endif

or

if *expression* **then**

statement

[[**elseif** *expression* **then**

statement]...]

else

statement

endif

Examples

```
if NOT SelGraph() then
    echo There are no graphs open
    exit all
endif

if length(val)==1 then
    echo {refs[idx]} {val}
else
    echo Duplicate reference {refs[idx]}. Ignoring
endif
if opts[0] && opts[1] then
    let sel = 1
elseif opts[0] then
    let sel = 2
else
    let sel = 3
endif
```

In form1, if the expression resolves to a TRUE value the statement will be executed. (TRUE means not zero, FALSE means zero). In the second form the same happens but if the expression is FALSE the statement after the **`else`** is executed. In the third form, if the first expression is FALSE, the expression after the **`elseif`** is tested. If that expression is TRUE the next statement is executed if not control continues to the next **`elseif`** or **`else`**.

While Statement

While statements are of the form:

`do while` *expression*

statement

`loop`

or alternatively

`while` *expression*

statement

`endwhile`

Example

```
do while GetOption(opt)<>'FALSE'
    let n = n+1
    let opt = 'LibFile' & (n+99)
loop
```

Both forms are equivalent.

In while loops the expression is evaluated and if it is TRUE the statement is executed. The expression is then tested again and the process repeated. When the expression is FALSE the loop is terminated and control passes to the statement following the **`endwhile`**.

For Statement

These are of the form:

`for` *variable=expression1* **`to`** *expression2* [**`step`** *constant*]

statement

`next` *variable*

Example

This finds the sum of all the values in `array`.

```
for idx=0 to length(array)-1
    let sum = sum + array[idx]
next idx
```

A for loop executes *statement* for values of *variable* starting at *expression1* and ending with *expression2*. Each time around the loop *variable* is incremented by *expression3* or if there is no step expression, by 1. If *expression2* starts off with a value less than *expression1*, *statement* will not be executed at all.

Script Statement

A script statement is a call to execute another script. Scripts are executed initially by typing their name at the command line (or if the script has .txt extension, the .txt can be omitted) or selecting a key or menu which is defined to do the same. Scripts can also be called from within scripts in which case the call is referred to as *script statement*. Note that a script may not call itself.

Exit Statement

There are four types:

```
exit while
exit for
exit script
exit all
```

`exit while` forces the innermost while loop to terminate immediately. Control will pass to the first statement after the terminating **`endwhile`** or **`loop`**.

`exit for` does the same for *for loops*.

`exit script` will force the current script to terminate. Control will pass to the statement following the call to the current script.

`exit all` will abort all script execution and control will return to the command line.

Accessing Simulation Data

Overview

When a simulation is run, a number of vectors (scalars for dc operating point) are created providing the node voltages and branch currents of the circuit. These are just like variables used in a script and can be accessed in the same way. There are however a number of differences from a normal variable. These are as follows:

- Simulation vectors are placed in their own *group*.
- They are usually attached to a *reference* vector.
- They usually have a *physical type* (e.g. Volts, Amps etc.)
- Some are *aliases*.

Each of these is described in the following sections.

Groups

All variables are organised into groups. When Pulsonix Spice first starts, there is only one called the *Global* group and all global variables are placed in it. (See "Scope of Variables. Global Variables"). When a script executes a new group is created for it and its own - local - variables are placed there. The group is destroyed when the script exits as are its variables.

Each time a simulation run is started a new group is created and the data generated by the analysis is placed in the group. Groups from earlier runs are not immediately destroyed so that results from earlier runs can be retrieved. By default, three simulation groups are kept at any time with the oldest being purged as new ones are created. A particular group can be prevented from being purged by selecting the **Graphs and Data** menu and **Keep Current Data Group**.

Groups provide a means of organising data especially simulation data and makes it possible to keep the results of old simulation runs.

All groups have a name. Simulation group names are related to the analysis being performed. E.g. transient analyses are always tran*n* where n is a number chosen to make the name unique.

Variables within a group may be accessed unambiguously by using their *fully qualified* name. This is of the form:

groupname:variable name

E.g. tran1:vout

The Current Group

At any time a single group is designated the *current* group. This is usually the group containing the most recent simulation data but may be changed by the user with the **Graphs and Data**|**Change Data Group**... menu or with the SetGroup command. If a variable name is used in an expression that is not local (created in a script) or global, the current group is searched for

it. So when the command "Plot vout" is executed if vout is not a local or global variable Pulsonix Spice will look for it in the current group.

You can view the variables in the current group with the Display command. Run a simulation and after it is completed type Display at the command line. A list of available variables from the simulation run will be displayed. Some of them will be "aliases".

The ':' prefix

If a variable name is prefixed with a colon it tells Pulsonix Spice to only search the current group for that name. Local or global variables of the same name will be ignored.

The colon prefix also has a side effect which makes it possible to access vectors created from numbered nodes. SPICE2 compatible netlists can only use numbers for their node (=net) names. Pulsonix Spice always creates simulation vectors with the same name as the nets. If the net name is a number, so is the variable name. It was stated earlier that variable names must begin with a non-numeric character but in fact this is only partly true. Variable names that start with a digit or indeed consist of only digits can be used provided they are always accessed with the ':' prefix and therefore must reside in the current group.

Collections

Collections were originally developed for versions 2.0 to 3.1 and were used for multi- step analyses such as Monte Carlo. The data for these analyses are now organised very differently using multi-division vectors described below.

Collections are still supported but are no longer used by the front end. They continue to be provided for backward compatibility but support for them may be withdrawn for future releases.

Multi-division Vectors

Multi-step runs such as Monte Carlo produce multiple vectors representing the same physical quantity. In Pulsonix Spice version 3.1 and earlier these vectors remained independent but the groups to which they were attached were bundled together into a collection. From version 4 the multiple vectors are in effect joined together into a multi-division vector. This is similar to a two dimensional vector (or array or matrix) except that the rows of the matrix are not necessarily all the same length.

When plotting a multi-division vector, each individual vector - or division - will be displayed as a single curve. If listing or printing a multi-division vector with the `Show` command, all the divisions will be listed separately.

You can access a single vector (or division) within a multi-division vector using the index operators - '[' and ']'. Suppose `VOUT` was a multi-division vector with 5 divisions. Each individual vector can be accessed using `VOUT[0]`, `VOUT[1]`, `VOUT[2]`, `VOUT[3]` and `VOUT[4]`. Each of these will behave exactly like a normal single division vector. So, you can use the index operator to access single elements e.g. `VOUT[2][23]` retrieves the single value at index 23 in division 2.

To find the number of divisions in a multi-division vector, use the function NumDivisions.

You can collate values at a given index across all divisions using the syntax: *vectorname*[][*index*]. E.g. in the above example `VOUT[][23]` will return a vector of length 5 containing the values of index 23 for all 5 divisions.

Multi-division vectors may be combined using arithmetic operators provided either both sides of the operator are compatible multi-division vectors - i.e. have identical x- values - or one of the values is a scalar.

Multi-division Vectors in Functions

Not all functions accept multi-division vectors for their arguments. The following table lists the functions that do accept multi-division vectors. The entry for each argument specifies whether that argument accepts multi-division vectors and how the data is dealt with.

“X”	Multi-division vectors are not accepted for this argument.
“Scalar”	The function acts on the multi-division vector to obtain a scalar value.

“Vector” The function obtains a scalar value for each division within the multi-division vector.

“Multi” The function processes all the vector's data to return a multi- division vector

Function name	Arg 1	Arg 2	Arg 3	Arg 4
abs	Multi			
atan	Multi			
atan_deg	Multi			
cos	Multi			
cos_deg	Multi			
db	Multi			
DefineFourierDialog	X	Scalar		
diff	Multi			
Execute	X	Multi	Multi	Multi
exp	Multi			
fft	Multi	X		
FIR	Multi	X	X	
Fourier	Multi	X	X	X
FourierOptionsDialog	X	Scalar		
FourierWindow	Multi	X	X	
GetVecStepParameter	Scalar			
GetVecStepVals	Scalar			
GroupDelay	Multi			
Histogram	Multi	X		
IIR	Multi	X	X	
im	Multi			
imag	Multi			
integ	Multi			
Interp	Multi	X	X	X
IsComplex	Scalar			
IsNum	Scalar			
IsStr	Scalar			
Length	Scalar			
ln	Multi			
log	Multi			
log10	Multi			
mag	Multi			
magnitude	Multi			
maxidx	Multi			
Maxima	Multi	X	X	
Maximum	Multi	X	X	
mean	Multi			
Mean1	Multi	X	X	
minidx	Multi			
Minima	Multi	X	X	
Minimum	Multi	X	X	
norm	Multi			
NumDivisions	Scalar			
NumElems	Vector			
ph	Multi			
phase	Multi			
phase_rad	Multi			
PhysType	Scalar			
Range	Multi	X	X	

re	Multi			
real	Multi			
Ref	Multi			
RefName	Scalar			
Rms	Multi			
RMS1	Multi	X	X	
rnd	Multi			
RootSumOfSquares	Multi	X	X	
sign	Multi			
sin	Multi			
sin_deg	Multi			
sqrt	Multi			
SumNoise	Multi	X	X	
tan	Multi			
tan_deg	Multi			
Truncate	Multi	X	X	
Units	Scalar			
Val	Multi			
XFromY	Multi	X	X	X
XY	Multi	Multi		
YFromX	Multi	X	X	

Vector References

Simulation vectors are usually attached to a *reference*. The reference is a vector's x-values. E.g. any vector created from a transient analysis simulation will have a reference of time. AC analysis results have a reference of Frequency.

Vectors created by other means may be assigned a reference using the **SetRef** command.

Physical Type

Simulation vectors also usually have a *Physical Type*. This identifies the values units e.g. Volts or Amps. When evaluating expressions Pulsonix Spice attempts to resolve the physical type of the result. For example, if a voltage is multiplied by a current, Pulsonix Spice will assign the Physical Type Watts to the result.

Any vector can be assigned a physical type using the **SetUnits** command.

User Interface to Scripts

Dialog Boxes

A number of functions are available which provide means of obtaining user input through dialog boxes. These are:

Function name	Page	Comment
AddRemoveDialog	39	Add or remove items to or from a list
BoolSelect	42	Up to 6 check boxes.
ChooseDir	44	Select a directory
EditSelect	54	Up to 6 edit boxes
EnterTextDialog	54	Enter multi line text
GetUserFile	82	Get file name (general purpose)
InputGraph	89	Input text for graph
RadioSelect	105	Up to 6 radio buttons
SelectDialog	113	Select item(s) from a list
TreeListDialog	121	Select item from tree structured list
UpDownDialog	124	Re order items
ValueDialog	124	Up to 10 edit boxes for entering values

The above are the general purpose user interface functions. There are many more specialised functions. These are listed in “Functions by Application”.

User Control of Execution

Sometimes it is desirable to have a script free run with actions controlled by a key or menu item. For example you may require the user to select an arbitrary number of curves on a graph and then press a key to continue operation of the script to perform - say - some calculations with those curves. You can use the **DefKey** and **DefMenu** commands to do this. However, for a key or menu to function while a script is executing, you must specify "immediate" mode when defining it. Only a few commands may be used in "immediate" mode definitions. To control script execution, the **Let** command may be used. The procedure is to have the key or menu assign a global variable a particular value which the script can test. The following example outputs messages if F2 or F3 is pressed, and aborts if F4 is pressed.

```
defkey F2 "scriptresume;let global:test=1" 5
defkey F3 "scriptresume;let global:test=2" 5
defkey F4 "scriptresume;let global:test=0" 5
let global:test = -1
while 1
    scriptpause
    if global:test=0 then
        exit script
    elseif global:test=1 then
        echo F2 pressed
    elseif global:test=2 then
        echo F3 pressed
    endif
    let global:test = -1
endwhile
unlet global:test
```

Errors

Loosely, there are two types of error, syntax errors and execution errors.

Syntax Errors

Syntax errors occur when the script presented deviates from the language rules. An **`endif`** missing from an *if statement* for example. Pulsonix Spice will attempt to find all syntax errors - it won't abort on the first one - but it will not execute the script unless the script is free of syntax errors. Sometimes one error can hide others so that fixing syntax errors can be an iterative process. On many occasions Pulsonix Spice can identify the details of the error but on some occasions it is unable to determine anything other than the fact that it isn't right. In this instance a "Bad Statement" error will be displayed. These are usually caused by unterminated *if, while* or *for* statements. Although in many cases Pulsonix Spice can correctly identify an unterminated statement, there are some situations where it can't.

Note that a syntax error in an expression will not be detected until execution.

Execution Errors

These occur when the script executes and are mostly the result of a command execution failure or an expression evaluation failure. Refer to error message documentation in the Help system for details of individual messages.

Error Messages

A listing of virtually all possible error messages is provided in the on-line help. This can be accessed from the **Help** menu and **Error Messages**.

Executing Scripts

Scripts are executed by typing their file name at the command line or selecting the menu File|Scripts|Run Script...Additionally, scripts can be assigned to a key or menu. See " User Defined Key and Menu Definitions".

If a full pathname is not given, the simulator first searches a number of locations. The rules are a little complicated and are as follows:

1. Search the BiScript directory followed by all its descendants. On Windows the BiScript directory is usually at <root>\support\biscript.
2. Search for a built in script of that name. Built in scripts are bound into the executable binary of the program. See "Built-in Scripts".
3. Search the SCRIPT directory. This is defined by the ScriptDir option which can also be accesses in the File Locations tab of the options dialog box. (File|Options|General...).
4. Search the User Script list of directories. This is defined by the UserScriptDir option variable. This may be set to a semi-colon delimited list of search paths.
5. Search the current working directory if the script was executed from a menu or the command line. If the script was called from another script, the directory where the calling script was located is searched instead

Scripts can also be executed using the Execute command.

Script Arguments

You can pass data to and from scripts using arguments.

Passing by Value

To pass a value *to* a script, simply place it after the script name. E.g.

```
my_script 10
```

The value 10 will be passed to the script. There are two methods of retrieving this value within the script. The easiest is to use the **Arguments** command. In the script you would place a line like:

```
Arguments num
```

In the above the variable `num` would be assigned the value 10. If the **Arguments** command is used, it becomes compulsory to pass the argument. If you wish to provide a script with optional arguments you must use the **$arg** variables. When an argument is passed to a script a variable with name **$arg*n*** is assigned with the value where ***n*** is the position of the argument on the command line starting at 1. To find out if the argument has been passed, use the **ExistVec** function. E.g.

```
if ExistVec('$arg1') then
    .. action if arg 1 passed
else
    .. action if arg 1 not passed
endif
```

Passing by Reference

When an argument is passed by value, the script in effect obtains a local copy of that data. If it subsequently modifies it, the original data in the calling script remains unchanged even if a variable name was used as the argument. The alternative is to *pass by reference* which provides a means of passing data back to the calling script. To pass by reference you must pass a variable prefixed with the '@' character. E.g.

```
Let var = 10
my_script @var
```

To retrieve the value in the called script we use the **Arguments** command as we did for passing by value but also prefix with '@'. E.g.

```
Arguments @var
Let var = 20
```

The above modifies `var` to 20 and this change will be passed back to the `var` in the calling script. In the above example we have used the same variable name `var` in both the called and calling scripts. This is not necessary, we have just done it for clarity. You can use any name you like in either script.

Optional arguments passed by reference work the same way as arguments passed by value except that instead of using the variable **$arg*n*** you must use **$var*n***. You do not need to use '@' when accessing arguments in this way. See the internal script **define_curve** for an example.

Important. There is currently a limitation that means you can't use an argument passed by reference directly in a braced substitution. E.g.

```
{var}
```

where `var` is an argument passed by reference will not work. Instead you can assign the value to a local variable first. This is a limitation of the current release and will be corrected in the future.

Passing Large Arrays

In many computer languages it is usually recommended that you pass large data items such as arrays by reference as passing by value involves making a fresh copy which is both time consuming and memory hungry. Passing by reference only passes the location of the data so is much more efficient. In the Pulsonix Spice script language, however, you can efficiently pass large arrays by value as it uses a technique known as "Copy on Write" that does not make a copy of the data unless it is actually modified.

Built-in Scripts

All the scripts needed for the standard user interface are actually built in to the executable file. The source of all of these is supplied on the CD in a scripts directory.

Debugging Scripts

To see display of commands executed

You can watch the script being executed line by line by typing at the command line before starting the script:

Set EchoOn

This will cause the text of each command executed to be displayed in the message window. When you have finished you cancel this mode with:

Unset EchoOn

To single step a script

Run the script by typing at the command line:

ScriptPause ; *scriptname*

where *scriptname* is the name of the script you wish to debug. To be useful it is suggested that you enable echo mode as described above. To single step through the script, press F2.

Note that ScriptPause only remains in effect for the first script. Subsequent scripts will execute normally.

To abort a currently executing script

Press escape key

To pause a currently executing script

Press shift-F2. Note that it is not possible to run other commands while a script is paused but you can single step through it using F2.

To resume a paused script

Press <Ctrl-F2>

Startup Script

The startup script is executed automatically each time the simulator is launched. By default it is called startup.sxscr but this name can be changed with in the options dialog box. (File|Options|General...). The startup file may reside in the script directory (defined by ScriptDir option variable) or in a user script directory (defined by UserScriptDir option variable).

The most common use for the startup script is to define custom menus and keys but any commands can be placed there.

To edit the startup script, select the File|Scripts|Edit Startup menu item.

Chapter 4. Function Reference

Unsupported Functions

A very small number of functions are designated as *unsupported*. These are usually functions we developed for internal use and are not used by the user interface. They are unsupported in so much as we will be unable to fix problems that you may encounter with them.

If you do use an unsupported function and it is useful to you, please tell technical support - by Email preferably. If a number of users find the function useful we will raise its status to supported.

abs

Arguments:

Type:	real/complex array
Description:	vector
Compulsory:	Yes
Default:	

Return type: real array

Returns absolute value or magnitude of argument. This function is identical to the mag() function.

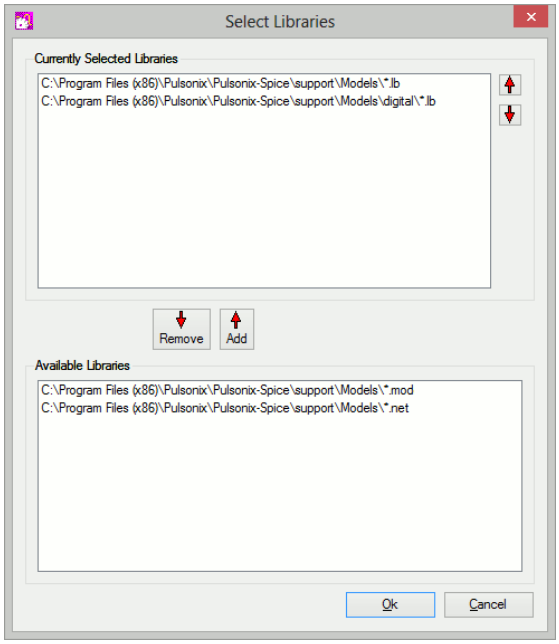
AddRemoveDialog

Arguments:

Type:	string array	string array	string array	string
Description:	Initial contents of selected list box	All items available	Options	Box style
Compulsory:	Yes	Yes	No	No
Default:			<<empty>>	'horizontal'

Return type: string array

Opens a dialog box to allow user to select from a number of items



The above shows the use of this function for adding and removing model libraries. The function is not, however, restricted to this application.

The function will display in the lower list box, all items found in both arguments 1 and arguments 2 with no duplicates. In the top list box, only the items found in argument 1 will be displayed. The user may freely move these items between the boxes. The function returns the contents of the top list box as an array of strings.

Argument 3 is a string array of size up to four, which may be used to specify a number of options. The first three are used for text messages and the fourth specifies a help topic to be called when the user presses the Help button. The help button will not be shown if the fourth element is empty or omitted.

0	Dialog box caption
1	Label for selected box
2	Label for available box
3	Help topic name

Argument 4 determines the style of the box. The above diagram shows the default (if arg 4 is absent, empty or 'horizontal'). If arg4 is set to 'vertical', the two list boxes will be arranged side by side instead of above each other

The function returns an empty vector if "Cancel" is selected.

arg

Type	real/complex
Description	vector
Compulsory	Yes
Default	

Return type: real

Returns the phase of the argument in degrees. Unlike the functions phase and phase_rad, this function wraps from 180 to -180 degrees. See arg_rad function below for a version that returns phase in radians.

arg_rad

Type	real/complex
Description	vector
Compulsory	Yes
Default	

Return type: real

Returns the phase of the argument in radians. Unlike the functions phase and phase_rad, this function wraps from p to -p radians. See arg function above for a version that returns phase in degrees.

Ascii

Type	string
Description	
Compulsory	Yes
Default	

Return type: real

Returns the ASCII code for the first letter of the argument

atan

Arguments:

Type:	real/complex
Description:	vector
Compulsory:	Yes
Default:	

Return type:real/complex array

Returns the arc tangent of its argument. If degrees option is set return value is in degrees otherwise radians.

atan_deg

Type	real/complex
Description	vector
Compulsory	Yes
Default	

Return type:real/complex array

Returns the arc tangent of its argument. Result is in degrees.

BoolSelect

Arguments:

Arguments:			
Type	real	string	string
Description	Initial check box settings	Labels	Dialog Box Caption
Compulsory	No	No	No
Default			
Arguments:			

Return type: real array

Opens a dialog box with up to 6 check boxes. The return value is a real vector containing the user's check box settings. 1 means checked, 0 means not checked. The number of check boxes displayed is the smaller of the length of arguments 1 and 2. If neither argument is supplied, 6 check boxes will be displayed without labels.

If the user cancels the operation, an empty value is returned. This can be checked with the length() function.

Example

The following dialog box is displayed after a call to:

```
BoolSelect([0,1,0], ['Label1', 'Label2', 'Label3'], 'Caption')
```



See Also

EditSelect

RadioSelect

ValueDialog

CanOpenFile

Type	string	string
Description	file name	options
Compulsory	Yes	No
Default		read

Return type: real

Returns TRUE (1) if file specified by argument 1 can be opened otherwise returns FALSE (0). Argument 2 may be set to ‘read’ (the default) or ‘write’ specifying what operation is required to be performed on the file.

This function takes account of lock files used to prevent other instances of Pulsonix from opening a file. For example, when a schematic is opened in non read only mode, a lock file is created which will prevent another instance of Pulsonix from opening that file but will not prevent another application from opening the file. CanOpenFile will return false for such files when ‘write’ mode is specified.

ChangeDir

Arguments:

Type:	string
Description:	New directory
Compulsory:	Yes
Default:	

Return type: real

Change current working directory to that specified by argument.

Return value is:

0: Success

1: Cannot create directory

2: Disk invalid

Char

Arguments:

Arguments:		
Type:	string	real
Description:	Input string	Character position
Compulsory:	Yes	Yes
Default:		

Return type: string

Returns a string consisting of the single character in arg1 located at index given in arg2. The first character has index 0. An empty string is returned if the index is out of range.

Example

```
Show Char('Hello World!', 4)
```

displays result:

```
Char('Hello World!', 4) = 'o'
```

ChooseDir

Arguments:

Type	string	string	string
Description	Starting directory	Dialog box caption	Message
Compulsory	No	No	No
Default	Current directory	'Choose Directory'	'Double-click directory to select'

Return type:string

Opens a dialog box showing a directory tree. Returns path selected by user or an empty string if cancelled. Initial directory shown specified in argument1.

CloseEchoFile

No arguments

Closes the file associated with the Echo command. For more information, see OpenEchoFile

CollectionName

Arguments:

Type:	string
Description:	Group name
Compulsory:	No
Default:	Current group

Return type: string

Returns the collection name of the group specified in the argument or current group if no argument given.

Collections were developed for collecting data groups created by multiple runs from version 1. From version 1.5, multiple run data is organised differently using “multi-division” vectors and consequently collections are largely obsolete. The collection functions and commands are still available but may not be supported in future versions. The plot and curve commands no longer support collections.

ComposeDigital

Type	string	real array	string array
Description	Bus name	Index range	Options
Compulsory	Yes	No	No
Default		See notes	

ComposeDigital builds a new vector from a binary weighted combination of digital vectors. It is intended to be used to plot or analyse digital bus signals. The simulator outputs bus signals as individual vectors. To plot a bus signal as a single value - either in numeric or analog form - these individual vectors must be combined as one to create a single value.

Note that ComposeDigital can only process purely digital signals. These are expected to have one of three values namely 0, 1 and 0.5 to represent an invalid or unknown state.

Argument 1

Signal root name. The function expects a range of vectors to be available of the form *busname#n. busname* is specified in argument 1 while the range of values for *n* is specified in argument 2.

Argument 2

Index range. The function processes vectors from *busname#idx_start* to *busname#idx_end*. *idx_start* and *idx_end* are specified by this argument as a two

dimensional array. For example if arg 1 is ‘BUS’ and arg 2 is [0,3], the function will process vectors:

BUS1#0

BUS1#1

BUS1#2

BUS1#3

as long all 4 vectors exist. If one or more vectors do not exist the first contiguous set of vectors will be used within the indexes specified. So if BUS1#0 didn’t exist, the function would use BUS1#1 to BUS1#3. If BUS1#2 didn’t exist, it would use just BUS1#0 and BUS1#1.

Note that the index may not be larger than 31.

Argument 3

1 or 2 element string array. Values may be any combination of ‘holdInvalid’ and ‘scale’.

‘holdInvalid’ determines how invalid states in the input are handled. If the ‘holdInvalid’ option is specified, they are treated as if they are not present and the previous valid value is used instead. If omitted, invalid states force an output that alternates between -1 or -2. This is to allow consecutive invalid states to be distinguished. For example, suppose there are 4 bits with one bit invalid. If one of the valid bits changes, the end result will still be invalid, but it sometimes desirable to know that the overall state has changed. So, in this case the first invalid state will show as a -1 and the second invalid state will be -2. In any following invalid state, the result will be -1 and so on.

scale’ forces the output to be scaled by the value $\mathbf{2}^{-(udxend - idxstart+1)}$

Return Value

The return value is a real vector that is the binary weighted sum of the vectors defined by arg 1 and arg 2 but treating invalid values (=0.5) as described above. So, in the example above, the result will be:

BUS1#0 + BUS1#1 x 2 + BUS1#2 x 4 + BUS1#3 x 8

CopyURL

Type	string	string	string
Description	From URL file	To URL file	options
Compulsory	Yes	Yes	No
Default			progress

Return Type: String array

Copies a file specified by a URL from one location to another. The URL may specify HTTP addresses (prefix 'http://'), FTP addresses (prefix 'ftp://') and local file system addresses (prefix 'file:/').

Argument 1

URL of source file.

Argument 2

URL of destination file

Argument 3

Options: can be 'progress' or 'noprogress'. If set to 'progress' (the default) a box will display with a bar showing the progress of the file transfer. Otherwise no such box will display.

Return Value

String array of length 2. First element will be one of the values shown in the following table:

Id	Description
UserAbort	User aborted operation
TimedOut	Connection timed out. This error usually occurs when an Internet connection is down.
NoError	Operation completed successfully
Unexpected1	This is an internal error that should not occur
UnknownProtocol	The protocol is unknown. I.e the URL prefix is not implemented. (Only HTTP, FTP and FILE are implemented)
Unsupported	This is an internal error that should not occur
ParseError	The URL does not comply with the expected format
IncorrectLogin	A username and password are required for this URL
HostNotFound	The specified host in the URL could not be found. This error can also occur if there is no Internet connection.
Unexpected2	This is an internal error that should not occur
MkdirError	Could not create target directory
RemoveError	This is an internal error that should not occur
RenameError	This is an internal error that should not occur
GetError	An error occurred while fetching a file
PutError	An error occurred while storing a file
FileNotExist	File doesn't exist
PermissionDenied	You do not have sufficient privilege to perform the operation
Unknown Error	This is an internal error that should not occur

The second element of the returned string gives a descriptive message providing more information about the cause of failure.

COS

Arguments:

Type:	real/complex array
Description:	vector
Compulsory:	Yes

Default:	

Return type: real/complex array

Return cosine of argument. Result is in radians.

cos_deg

Type	real/complex array
Description	vector
Compulsory	Yes
Default	

Return type: real/complex array

Return cosine of argument. Result is in degrees.

CreateShortcut

Type	string	string	string
Description	Path of object	Path of link file	Description
Compulsory	Yes	Yes	Yes
Default			

Return type: string

This function is only available on the Windows platform.

Create a ‘shortcut’ to a file or directory.

Argument 1

Path of file or directory which shortcut will point to

Argument 2

Path of shortcut itself.

Argument 3

Description of shortcut

Return Value

‘Success’ or ‘Fail’

Date

Type	string
Description	format
Compulsory	No
Default	‘locale’

Return type: string

Returns the current date in the format specified.

Argument 1

May be ‘iso’ or ‘locale’. When set to ‘locale’ the date is returned in a format specified by system settings. When set to ‘iso’ the date is returned in a format complying with ISO8601 which is YYYY-MM-DD where YYYY is the year, MM is the month of the year (between 01 and 12), and DD is the day of the month between 01 and 31.

dB

Arguments:

Type:	real/complex array
Description:	vector
Compulsory:	Yes
Default:	

Return type: real/complex array

Returns 20*log10(mag(argument))

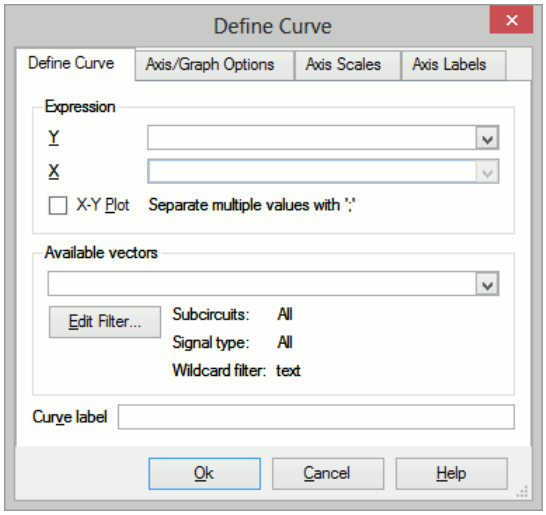
DefineCurveDialog

Arguments:

Type:	string array
Description:	initial values
Compulsory:	Yes
Default:	

Return type: string array

Opens the following dialog box to define a curve for plotting



The argument is a string array of length 25 which defines how the various controls are initialised. This array has the same format for EditAxisDialog. Not all the elements are relevant to this function. The following table describes the elements that are used.

Index	Purpose	Notes	Default
0	Axis Type	Setting of "Axis Type" group in "Axis/Graph Options sheet". Possible values: 'auto' "Auto Select" 'selected' "Use Selected" 'axis' "Use New Y-Axis" 'grid' "Use New Grid" 'digital' "Digital"	No default. Value must be specified.
1	Graph Type	Setting of "Graph Options" group in "Axis/Graph Options sheet". Possible values: 'add' "Add To Selected" 'newsheet' "New Graph Sheet" 'newwindow' "New Graph Window"	No default. Value must be specified.
2	Axis name	Not used with this function	
3	Persistence	Not used with this function	
4	Graph name	Not used with this function	
5	Curve label	"Curve label" control in "Define Curve" sheet	<<empty>>

6	Analysis	Not used with this function	
7	Plot on completion	Not used with this function	
8	reserved for future use	Not used with this function	
9	reserved for future use	Not used with this function	
10	X axis graduation	Setting of Log|Lin|Auto for X Axis in "Axis Scales" sheet. Possible values: 'lin' "Lin" 'log' "Log" 'auto' "Auto"	'auto'
11	X axis scale options	Setting of scale options for X Axis in "Axis Scales" sheet. Possible values: 'nochange' "No Change" 'auto' "Auto scale" 'defined' "Defined"	'auto'
12	Y axis graduation	Setting of Log|Lin|Auto for Y Axis in "Axis Scales" sheet. Possible values as for X axis.	'auto'
13	Y axis scale options	Setting of scale options for X Axis in "Axis Scales" sheet. Possible values as for X axis.	'auto'
14	X axis min limit	Min value for X Axis in "Axis Scales" sheet. Must be specified as a string.	0
15	X axis max limit	Max value for X Axis in "Axis Scales" sheet. Must be specified as a string.	1
16	Y axis min limit	Min value for Y Axis in "Axis Scales" sheet. Must be specified as a string.	0
17	Y axis max limit	Max value for Y Axis in "Axis Scales" sheet. Must be specified as a string.	1
18	X axis label	"Axis Label" setting for "X-Axis" group in "Axis Labels" sheet	<<empty>>
19	X axis units	"Axis Units" setting for "X-Axis" group in "Axis Labels" sheet	<<empty>>
20	Y axis label	"Axis Label" setting for "Y-Axis" group in "Axis Labels" sheet	<<empty>>
21	Y axis units	"Axis Units" setting for "Y-Axis" group in "Axis Labels" sheet	<<empty>>
22	Y-expression	Contents of Y expression edit box	<<empty>>
23	X-expression	Contents of X expression edit box, if enabled	<<empty>>
24	Vector filter	Subcircuit filter selection in "Available Vectors" group. Possible values 'all' "All" 'top' "Top level" sub circuit name. Select a subcircuit name.	

The available vectors list box is initialised with the names of vectors in the current group.

The function returns a string array with the same format as the argument. If the user selects "Cancel" the function returns an empty vector.

DescendDirectories

Type	string
Description	Directory
Compulsory	Yes
Default	

Return type: string array

Returns all directories under the specified directory. DescendDirectories recurses through all sub-directories including those pointed to by symbolic links. DescendDirectories only returns directory names. It does not return files. Use the ListDirectory function to return the files in a directory.

diff

Arguments:

Type:	real array
Description:	vector
Compulsory:	Yes
Default:	

Return type: real array

Returns the derivative of the argument with respect to its reference. If the argument has no reference the function returns the derivative with respect to the argument's index - in effect a vector containing the difference between successive values in the argument. For details on references see “Vector References”.

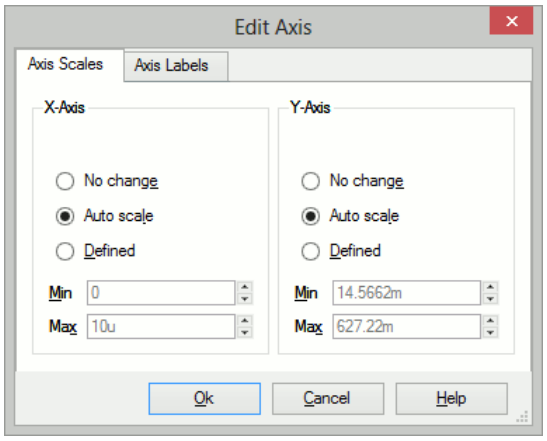
EditAxisDialog

Arguments:

Type:	string array
Description:	initial settings
Compulsory:	Yes
Default:	

Return type: string array

Opens the following dialog box used to edit graph axes



The argument is a string array of length 25 which defines how the various controls are initialised. This array has the same format for DefineCurveDialog. Not all the elements are relevant to this function. The following table describes the elements that are used.

0	Axis Type	Not used with this function
1	Graph Type	Not used with this function
2	Axis name	Not used with this function
3	Persistence	Not used with this function
4	Graph name	Not used with this function
5	Curve label	Not used with this function
6	Analysis	Not used with this function
7	Plot on completion	Not used with this function
8	reserved for future use	Not used with this function
9	reserved for future use	Not used with this function
10	X axis graduation	Not used with this function

11	X axis scale options	Setting of scale options for X Axis in "Axis Scales" sheet. Possible values: 'nochange' "No Change" 'auto' "Auto scale" 'defined' "Defined"	'auto'
12	Y axis graduation	Not used with this function	
13	Y axis scale options	Setting of scale options for X Axis in "Axis Scales" sheet. Possible values as for X axis.	'auto'
14	X axis min limit	Min value for X Axis in "Axis Scales" sheet. Must be specified as a string.	0
15	X axis max limit	Max value for X Axis in "Axis Scales" sheet. Must be specified as a string.	1
16	Y axis min limit	Min value for Y Axis in "Axis Scales" sheet. Must be specified as a string.	0
17	Y axis max limit	Max value for Y Axis in "Axis Scales" sheet. Must be specified as a string.	1
18	X axis label	"Axis Label" setting for "X-Axis" group in "Axis Labels" sheet	<<empty>>
19	X axis units	"Axis Units" setting for "X-Axis" group in "Axis Labels" sheet	<<empty>>
20	Y axis label	"Axis Label" setting for "Y-Axis" group in "Axis Labels" sheet	<<empty>>
21	Y axis units	"Axis Units" setting for "Y-Axis" group in "Axis Labels" sheet	<<empty>>
22	Y-expression	Not used with this function	
23	X-expression	Not used with this function	
24	Vector filter	Not used with this function	

The function returns a string array with the same format as the argument. If the user selects "Cancel" the function returns an empty vector.

EditCrosshairDimensionDialog

Type	string array	string array	string array
Description	Property names	Property values	Property types
Compulsory	Yes	Yes	No
Default			

Return type:string array

Opens a dialog intended for editing the characteristics of cursor crosshair dimensions.

The Properties sheet behaves in the same way as the EditObjectPropertiesDialog and is initialised by the function's arguments. The Edit sheet allows the edit and display of certain properties as defined in the following table:

Property Name	**Affects Control:**
Label1	Label 1
Label2	Label 2
Label3	Label 3
Style	Contents of Style box. One of six values: Auto: Automatic, Show Difference Internal Internal, Show Difference External External, Show Difference P2P1 Show Absolute P2P1Auto Automatic, Show Difference, Show Absolute None No controls selected
Font	Font. String defining font specification

If any of the controls in the **Edit** sheet are changed, the corresponding property values in the **Properties** sheet will reflect those changes and vice-versa.

EditCurveMarkerDialog

Type	string array	string array	string array
Description	Property names	Property values	Property types
Compulsory	Yes	Yes	No
Default			

Return type:string array

Opens a dialog intended for editing the characteristics of curve markers.

The Properties sheet behaves in the same way as the EditObjectPropertiesDialog and is initialised by the functions arguments. The Edit sheet allows the edit and display of certain properties as defined in the following table:

Property Name	Affects Control:
Label	Label
LabelJustification	Text Alignment box. One of these values: -1 Automatic 0 Left-Bottom 1 Centre-Bottom 2 Right-Bottom 3 Left-Middle 4 Centre-Middle 5 Right-Middle 6 Left-Top 7 Centre-Top 8 Right-Top
Font	Font. String defining font specification

If any of the controls in the Edit sheet are changed, the corresponding property values in the Properties sheet will reflect those changes and vice-versa.

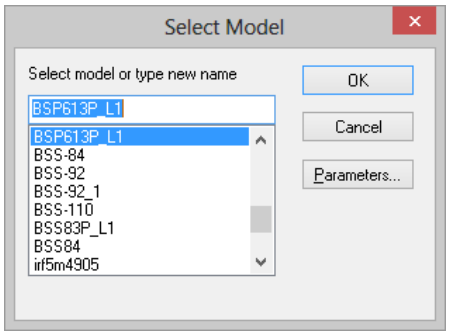
EditDeviceDialog

Arguments:

Type:	string array	string array	string array	string array
Description:	options/initial settings	devices	parameter names	parameter values
Compulsory:	Yes	Yes	No	No
Default:			<<empty>>	<<empty>>

Return type: string array

Opens the following dialog box used to select a device and optionally specify its parameters.



Argument 1

Defines options and initial settings as follows:

0	Text entered in edit control above list box. If the text item is also present in the device list (argument 2), then that item will be selected.
1	Ignored unless element 1 is empty. Integer (entered in string form) which defines selected device.
2	Dialog box caption. Default if omitted: "Select Device"

3	Message at the top of the dialog box. . Default if omitted: "Select Device"

Argument 2

String array defining the list of devices.

Argument 3

String array defining list of parameter names. See argument 4.

Argument 4

String array defining list of parameter values. If arguments 3 and 4 are supplied the "Parameters..." button will be visible. This button opens another dialog box that provides the facility to edit these parameters' values.

Return value

The function returns a string array in the following form

0	Entry in the text edit box.
1	Index into device list (argument 2) of device in text edit box. If this device is not in the list, -1 will be returned.
2	Number of parameter values.
3 onwards	The values of the parameters in the order they were passed.

If the user selects "Cancel" the function returns an empty vector.

EditFreeTextDialog

Type	string array	string array	string array
Description	Property names	Property values	Property types
Compulsory	Yes	Yes	No
Default			

Return type: string array

This function is almost identical to the EditCurveMarkerDialog functions except for some changes to the aesthetics of the dialog box.

EditGraphTextBoxDialog

Type	string array	string array	string array
Description	Property names	Property values	Property types
Compulsory	Yes	Yes	No
Default			

Return type: string array

Opens a dialog intended for editing the characteristics of text box objects for graphs.

The Properties sheet behaves in the same way as EditObjectPropertiesDialog and is initialised by the function's arguments. The Edit sheet shown above allows the edit and display of certain properties as defined in the following table:

Property Name	Affects Control:
Label	Label
Colour	Background Colour. An integer defining the RGB value
Font	Font. String defining font specification

If any of the controls in the Edit sheet are changed, the corresponding property values in the Properties sheet will reflect those changes and vice-versa.

EditLegendBoxDialog

Type	string array	string array	string array
Description	Property names	Property values	Property types
Compulsory	Yes	Yes	No

Default

Return type: string array

This function is virtually identical to EditGraphTextBoxDialog above except for a different caption.

EditSelect

Arguments:

Type:	string	string	string
Description:	initial edit control entries	labels	dialog box caption
Compulsory:	No	No	No
Default:	<<empty>>	<<empty>>	<<empty>>

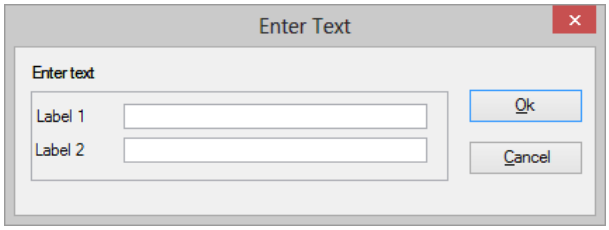
Return type: string array

Opens a dialog box containing up to 6 *edit controls* allowing the user to enter text values. The number of edit controls is the smaller of the lengths of arguments 1 and 2. If no arguments are given, 6 controls will be displayed with blank labels. Function returns string vectors containing user entries for each control. If cancel is selected, a single empty string is returned.

Example

The following dialog box will be displayed on a call to

```
EditSelect(['Init 1','Init 2'],['Label 1','Label 2'],'Enter Text')
```



See Also

BoolSelect

RadioSelect

ValueDialog

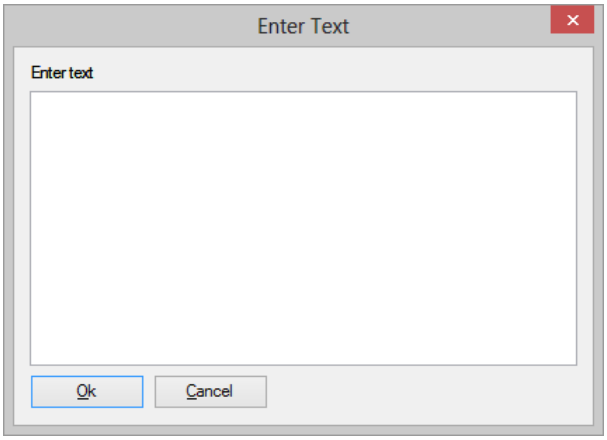
EnterTextDialog

Arguments:

Type:	string array
Description:	initial text and box caption
Compulsory:	Yes
Default:	

Return type: string array

Opens the following, allowing the user to enter lines of text.



The argument specifies the initial text and the dialog box's caption as follows:

0	Initial text
1	Dialog box caption

The function returns the text entered by the user.

Execute

Type	string	any	**May have up to 8 args in total**
Description	Script name	Script argument 1	Script args 2 - 7
Compulsory	Yes	No	No
Default			

Return type: Depends on called script

Function calls the script defined in arg 1 and passes it the arguments supplied in arg 2- 8. The function's returned value is the script's first argument passed by reference.

The Execute function is used internally to implement user functions that are registered with the RegisterUserFunction command.

ExistDir

Arguments:

Type:	string
Description:	directory name
Compulsory:	Yes
Default:	

Return type: real

Function returns a real scalar with one of three values:

0	Directory does not exist
1	Directory exists but with no write privilege
2	Directory exists with write privilege

ExistFunction

Type	string	string
Description	Function name	Function type
Compulsory	Yes	No
Default		'global'

Return type: real

Returns TRUE or FALSE depending on whether specified function exists.

Argument 1

Function name.

Argument 2

Either ‘global’ or ‘script’. If ‘global’, arg 1 is assumed to be the name of a built in function. If ‘script’ arg 1 is assumed to be a function defined as a script and installed using the RegisterUserFunction.

User defined compiled functions linked in as a DLL/shared library are treated as ‘global’.

ExistVec

Arguments:

Type:	string	string
Description:	vector name	options
Compulsory:	Yes	No
Default:		

Return type: real

Returns TRUE (1) if the specified vector exists otherwise returns FALSE (0). If the second argument is 'GlobalLocal', only the global and local groups are searched for the vector otherwise the current group is also searched. See Accessing Simulation Data for details on groups.

EXP

Arguments:

Type:	real/complex array
Description:	vector
Compulsory:	Yes
Default:	

Return type: real/complex array

Returns e raised to the power of argument. If the argument is greater than 709.016, an overflow error occurs.

FFT

Arguments:

Type:	real array	string
Description:	vector	window function
Compulsory:	Yes	No
Default:		'Hanning'

Return type: complex array

Performs a **Fast Fourier Transform** on supplied vector. The number of points used is the next binary power higher than the length of argument1. Excess points are zero-filled. Window used may be 'Hanning' (default) or 'None'.

User's should note that using this function applied to raw transient analysis data will not produce meaningful results as the values are unevenly spaced. If you apply this function to simulation data, you must either specify that the simulator outputs at fixed intervals (select the **Output at interval** option in the Pulsonix Schematics **Simulation Parameters...** dialog

box) or you must interpolate the results using the **Interp** function (The FFT plotting menu items run a script which interpolate the data if it detects that the results are unevenly spaced. Use of these menus does not require special consideration by the user.)

The **fft** function described here never directly interpolates. Interpolation may be provided as a separate function (**Interp**) if required.

Note that much better results are obtained if the original data is genuinely evenly spaced and not interpolated. If the fixed interval option is specified for transient analysis, the simulator will actually perform an analysis at the evenly spaced points and good FFT results can be obtained. This is not the case with most other SPICE products which interpolate. Interpolation, introduces errors into FFT results especially at frequencies far removed from fundamental components.

Further information on FFT's can be found in the Pulsonix Spice User's manual.

Field

Type	real	real	real
Description	Value	first bit	second bit
Compulsory	Yes	Yes	Yes
Default			

Return type: real

Function provides bit access to integers. Returns the decimal value of a binary number composed from the binary representation of argument 1 between the bit numbers defined in arguments 2 and 3. E.g.:

Field(100, 1, 3) = 2

100 (decimal) = 1100100 (binary)

```
bits 1 to 3 (from right i.e. least significant) = 010 (binary) = 2
```

Field is useful for cracking the individual bits used for symbol attribute flags.

FindModel

Arguments:

Type:	string	string
Description:	model name	model type (e.g. 'Q' for BJT, 'X' for subcircuit)
Compulsory:	Yes	Yes
Default:		

Return type: string array

Returns string array of length 2 holding the file name and line number of the definition of the specified model.

FIR

Arguments:

Type:	real array	real array	real array
Description:	vector to be filtered	filter coefficients	initial conditions
Compulsory:	Yes	Yes	No
Default:			All zero

Return type: real array

Performs "Finite Impulse Response" digital filtering on supplied vector. This function performs the operation:

$y_n = x_n . c_0 + x_{n-1} . c_1 + x_{n-2} . c_2 ...$

Where:

x is the input vector (argument 1)

c is the coefficient vector (argument 2)

y is the result (returned value)

The third argument provide the "history" of x i.e. x_{-1}, x_{-2} etc. as required.

The operation of this function (and also the **IIR** function) is simple but its application can be the subject of several volumes! Below is the simple case of a four sample rolling average. In principle an almost unlimited range of FIR filtering operations may be performed using this function. Any text on Digital Signal Processing will provide further details.

User's should note that using this function applied to raw transient analysis data will not produce meaningful results as the values are unevenly spaced. If you apply this function to simulation data, you must either specify that the simulator outputs at fixed intervals (select the **Output at interval** option in the Pulsonix Schematics **Simulator Parameters...** dialog box) or you must interpolate the results using the **Interp** function.

Example

Suppose a vector VOUT exist in the current group (simulation results). The following will plot VOUT with a 4 sample rolling average applied

```
Plot FIR(vout, [0.25, 0.25, 0.25, 0.25])
```

Alternatively, the following does the same

```
Plot FIR(vout, 0.25*unitvec(4))
```

Floor

Arguments:

Type:	real
Description:	scalar
Compulsory:	Yes
Default:	

Return type: real

Returns the argument truncated to the next lowest integer. Examples

Floor(3.45) = 3

```
Floor(7.89) = 7
Floor(-3.45) = -4
```

FormatNumber

Arguments:

Type:	real	real	string
Description:	number	significant digits	format
Compulsory:	Yes	Yes	No
Default:			'eng'

Return type: real

Formats a real value and returns a string representation of it. Argument 2 is the number of significant digits and argument 3 specify what format to use. The options are:

'eng'	(default if omitted). Formats the number using engineering units
'noeng'	Normal format. Will use 'E' if necessary
'%'	Formats as a percentage

Fourier

Type	real array	real	real	real array
Description	data	Fundamental frequency	Number of frequency terms	options
Compulsory	Yes	Yes	Yes	No
Default				

Return type: complex array

Calculates the fourier spectrum of the data in argument 1. The function uses the 'Continuous Fourier' technique which numerically integrates the Fourier integral. Because this technique does not require the input data to be sampled at evenly spaced points, it doesn't suffer from frequency aliasing. This is the main drawback of the more commonly used FFT (Fast Fourier Transform) algorithm. However, the Continuous Fourier algorithm is much slower then the FFT, sometimes dramatically so.

Argument 1

The input data. This is expected to possess a reference i.e. x-values

Argument 2

Specifies the fundamental frequency. All terms calculated will be an integral multiple of this.

Argument 3

Specifies the number of frequency terms to be calculated.

Argument 4

This is optional and can be a 1 or 2 element array. The first element is the first frequency to be calculated expressed as a multiple of the fundamental. The default value is 0 i.e. the DC term is calculated first. The second element is the integration order used and may be 1 or 2.

Return Value

The result of the calculation and will be a complex array with length equal to argument 3.

FourierOptionsDialog

Type	string array	real array
Description	initial values	sample vector
Compulsory	Yes	No
Default		

Return type: string array

Same as DefineFourierDialog except that only the Fourier sheet is displayed. The remaining tabbed sheets are hidden.

FourierWindow

Type	real	string
Description	Input vector	window type
Compulsory	Yes	No
Default		'hanning'

Return type: real array

Returns the input vector multiplied by one of a selection of 4 window functions. This is intended to be used with a Fourier transform algorithm.

Argument 1

Input vector

Argument 2

Window type. One of:

'hanning'

'hamming'

'blackman'

'rectangular'

FullPath

Arguments:

Type:	string	string
Description:	relative path name	reference directory
Compulsory:	Yes	No
Default:		Current working directory

Return type: real

Returns the full path name of the specified relative path and reference directory.

Examples

```
FullPath('amplifier.net', 'c:\simulation\circuits') =
c:\simulation\circuits\amplifier.net

FullPath('..\amplifier.net', 'c:\simulation\circuits') =
c:\simulation\amplifier.net
```

See also

RelativePath

SplitPath

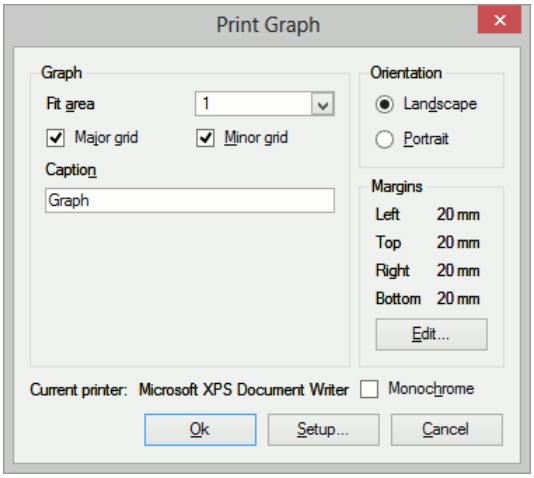
GenPrintDialog

Arguments:

Type:	string array
Description:	initial settings
Compulsory:	Yes
Default:	

Return type: string array

Opens the following dialog box used to define print settings



The arguments is a string array of length 13 and defines the initial settings of the dialog box as follows:

0	'area' "Fit Area" 'grid' "Fixed Grid"
1	Not used. Set to '1'
2	Not used. Set to ''
3	Graph magnification (entered as a string)
4	Graph caption
5	Orientation 'landscape' or 'portrait'
6	Layout: use '1' meaning Graph only
7	Left margin. The value is entered and returned in units of 0.1mm but will be displayed according to system regional settings. Must be entered as a string
8	Top margin. Comments as for left margin.
9	Right margin. Comments as for left margin.
10	Bottom margin. Comments as for left margin.
11	Major grid checked: 'on' Checked 'off' Not checked
12	Minor grid checked: 'on' Checked 'off' Not checked

The function returns a string array with the same format as the argument and assigned with the user's settings. If the user selects "Cancel" the function returns an empty vector.

GetAllCurves

No Arguments

Return type: string array

Returns an array listing id's for all curves on currently selected graph. All curves are referred to by a unique value that is the "id". Some functions and command require a curve id as an argument.

GetAllYAxes

No Arguments

Return type: string array

Returns an array listing all y axis id's for currently selected graph. All graph axes have a unique "id" which may be used with some other commands and functions.

GetAnalysisInfo

Type	string
Description	Options
Compulsory	No
Default	

Return type: string array

Returns the parameters of the most recent analysis performed by the simulator. The parameters are returned in the form of a string array. If argument 1 is set to 'name' the function will return the names of each parameter.

The following sample shows how to obtain a the stop time of a transient analysis:

```
let stopIdx = Search(GetAnalysisInfo('name'), 'tstop')

Let stopTime = Val(info[stopIdx])
```

The following table shows the parameter names currently available for each analysis type:

Analysis Type	Parameter Names
Transient	ANALYSISNAME, GROUPNAME, TSTART, TSTOP, TSTEP, TMAX, UIC, DELTA, RTNSTART, RTNSTOP, RTNSTEP, RTNENABLED, FAST
AC	ANALYSISNAME, GROUPNAME, PARAM, MODEL, TEMP, FREQ, MONTE, REPEAT, DEVICE, MODE, START, STOP, STEP, NUMSTEPS, GRAD, SINGLE, F
DC	ANALYSISNAME, GROUPNAME, PARAM, MODEL, TEMP, FREQ, MONTE, REPEAT, DEVICE, MODE, START, STOP, STEP, NUMSTEPS, GRAD, SINGLE
Noise	ANALYSISNAME, GROUPNAME, PARAM, MODEL, TEMP, FREQ, MONTE, REPEAT, DEVICE, MODE, START, STOP, STEP, NUMSTEPS, GRAD, SINGLE, V, VN, INSRC, PTSPERSUM, F
Transfer Function	ANALYSISNAME, GROUPNAME, PARAM, MODEL, TEMP, FREQ, MONTE, REPEAT, DEVICE, MODE, START, STOP, STEP, NUMSTEPS, GRAD, SINGLE, V, VN, I, INSRC, F, IMODE
Sensitivity	ANALYSISNAME, GROUPNAME, POSNAME, NEGNAME, I, GRAD, START, STOP, NUMSTEPS
Pole-zero	ANALYSISNAME, GROUPNAME, NODEINAME, NODEGNAME, NODEJNAME, NODEKNAME, VOLCUR, POLZER
Operating point	ANALYSISNAME, GROUPNAME

GetAxisCurves

Arguments:

Type:	string
Description:	Y axis id
Compulsory:	Yes
Default:	

Return type: string array

Returns an array listing all curve id's for specified y-axis. All curves are referred to by a unique value that is the "id". Some functions and command require a curve id as an argument.

GetAxisLimits

Arguments:

Type:	string
Description:	Axis id
Compulsory:	Yes
Default:	

Return type: string array

Returns array of length 3 providing limits info for specified axis as follows:

0	Minimum limit
1	Maximum limit
2	Axis scale type - 0 = linear, 1 = logarithmic
3	Fixed or auto. 0 = fixed, 1 = auto

GetAxisType

Arguments:

Type:	string
Description:	Axis id
Compulsory:	Yes
Default:	

Return type: string array

Returns string specifying type of axis. Possible values are:

'X'	X-axis
'Digital'	A Digital Y-axis. (Created with "Curve /dig")
'Main'	Main Y-axis (axes at bottom of graph)
'Grid'	Grid Y-axis (axes stacked on top of main)
'NotExist'	Axis does not exist

GetAxisUnits

Arguments:

Type:	string
Description:	Axis id
Compulsory:	Yes
Default:	

Return type: string array

Returns physical units of axis. See **Units** function for list of possible values.

GetColours

No arguments

Return type: string array

Returns the names of all objects in the program whose colour may be edited. The function is usually used in conjunction the GetColourSpec function, the SelectColourDialog function and the EditColour command.

GetColourSpec

Type	string
Description	Colour name
Compulsory	Yes
Default	

Return type: string

Returns the current colour specification for the object whose name is passed to argument 1. Valid object names can be obtained from the GetColours function. The return value may be used to initialise the SelectColourDialog which allows the user to define a new colour.

The return value represents the colour of the object as a single integer which can be decoded into its RGB components. However, this value should not be used directly as its format may change in future versions of the product.

If the object name passed is not recognised the function will return the representation for the colour black.

GetConfigLoc

No arguments

Return type: string array

Returns the location of the application's configuration settings. In versions prior to version 2, this would be in one of the following forms:REG;*registry_root_pathname*

OR

PATH;*inifile_pathname*

If the first form is returned, the settings are stored in the registry the path being HKEY_CURRENT_USER*registry_root_pathname*

If the second form is returned the settings are stored in a file with full path equal to *inifile_pathname*.

From version 2, the registry is no longer used for storing settings, so only the second of the two forms will ever be returned.

The return value from GetConfigLoc can be used directly as the value of the /config_location switch at the simulator (SIM.EXE) command line. See the “Running the Simulator” chapter in the *Simulator Reference Manual* for more details.

GetConvergenceInfo

No arguments

Return type: string array

Returns a string array providing convergence information about the most recent run. Each element of the array is a list of values separated by semi-colons. The output may be pasted into a spreadsheet program that has been set up to interpret a semicolon as a column separator. The first element of the array lists the names for each column and therefore provides a heading. The following headings are currently in use:

type	Node or Device
name	Name of node or device that failed to converge
count	Number of times node/device failed to converge during run
time (first step)	Time of most recent occurrence of a ‘first step’ failure.
required tol	Required tolerance for most recent ‘first step’ failure
actual tol	Tolerance actually achieved for most recent ‘first step’ failure
absolute val	Absolute value for most recent ‘first step’ failure
time (cut back step)	Time of most recent occurrence of a ‘cut back step’ failure.
required tol	Required tolerance for most recent ‘cut back step’ failure
actual tol	Tolerance actually achieved for most recent ‘cut back step’ failure
absolute val	Absolute value for most recent ‘cut back step’ failure
final?	Node or device failed on the final step that caused the simulation to abort
top analysis	Main analysis mode (Tran, DC etc.)
current analysis	Current analysis. Either the same as ‘top analysis’ or Op
op mode	Method being used for operating point. (PTA, JI2, GMIN or SOURCE)

A *first step* failure is a failure that occurred at the first attempt at a time step after a previously successful step. If a time point fails, the time step is cut back and further iterations are made. Failures on steps that have been cut back are referred to in the above table as *cut back steps*. Quite often the nodes or devices that fail on a *cut back step* are quite different from the nodes or devices that fail on a *first step*. The root cause of a convergence failure will usually be at the nodes or devices that fail on a *first step*.

It is quite difficult to interpret the information provided by this function. The ‘where’ script performs a simple analysis and sometimes displays the nodes or devices most likely to be the cause.

GetCurDir

No Arguments

Return type: string

Returns current working directory.

GetCurrentGraph

No arguments

Return type: string

Returns id of the currently selected graph. Returns '-1' if no graphs are open. The id can be used in a number of functions that return information about graphs or graph objects generally.

GetCursorCurve

No arguments

Return type: string

Returns a string array of length 3 providing information on the curve attached to the measurement cursor. Returns an empty vector if cursors not enabled.

Index	Description
0	Curve id
1	Source group name. This is the group that was current when the curve was created.
2	Division index if curve is grouped. (E.g. for Monte Carlo)

GetCurveAxis

Arguments:

Type:	string
Description:	curve id
Compulsory:	Yes
Default:	

Return type: string

GetCurveName

Arguments:

Type:	string
Description:	curve id
Compulsory:	Yes
Default:	

Return type: string

Returns name of specified curve

GetCurves

No Arguments

Return type: string array

Returns an array of curve names (as displayed on the graph legend) for the current graph.

GetCurveVector

Type	real	real	string
Description	curve id	Division index	Obsolete - no longer used
Compulsory	Yes	No	No
Default		Return all divisions	

Return type: real array

Returns the data for a curve.

For a single curve (i.e. not a group of curves as created from a Monte Carlo plot) only the first argument is required and this specifies the curve's id.

If the curve id refers to a group of curves created by a multi-step run, then the second argument may be used to identify a single curve within the group. The data for the complete curve set is arranged as a *Multi Division Vector*. The second argument specifies the division index. If absent the entire vector is returned

Note that the arguments to this function for version 4 and later have changed from earlier versions.

GetDatumCurve

No arguments

Return type: string array

Returns a string array of length 3 providing information on the curve attached to the reference cursor.

Index	Description
0	Curve id
1	Source group name. This is the group that was current when the curve was created.
2	Division index if curve is grouped. (E.g. for Monte Carlo)

GetDeviceDefinition

Type	string	string
Description	Device name	Device type
Compulsory	Yes	Yes
Default		

Return type: string array

Searches for the specified device model in the global library and returns the text of the model definition. If the device is defined using a .MODEL control, the result will have a single element containing the whole definition. If the device is defined using a subcircuit then the result will be a string array with a single element for each line in the subcircuit definition.

Argument 1

The model/subcircuit name. E.g. 'Q2N2222' or 'TL072'

Argument 2

The type of the device. This may be either the device letter e.g. 'Q' for a BJT, or the model type name e.g. 'npn'. A list of device letters is given in the *Simulator Reference manual* in the “Running the Simulator” chapter.

If the device is a subcircuit, use the letter 'X'.

GetDeviceInfo

Type	string	string
Description	Model name	Options
Compulsory	Yes	No
Default		none

Return type: string array

Returns information about the specified simulator device.

Argument 1

Internal device name as returned by the GetModelType or GetInternalDeviceName function. This is not the same as the type name used in the .MODEL control but a name that is used internally by the simulator. For example, the internal device name for a LEVEL 1 MOSFET is 'MOS1'.

Optionally the device letter may be specified if arg2 = 'letter'. However, the function will not return such precise information if this option is used. For example, the LEVEL value will not be known and so -1 will be returned. Also the minimum and maximum number of terminals will reflect all devices that use that device letter and not just one specific device. E.g. the ‘BJT’ device defines the standard SPICE Gummel-Poon transistor which can have 3 or 4 terminals. But the ‘q’ letter can also specify VBIC_Thermal devices which can have 5 terminals.

Argument 2

Options, currently only one. If this is set to 'letter', a single letter should be specified for argument 1. This is the device letter as used in the netlist, e.g. 'Q' for a BJT, 'R' for a resistor. See notes above concerning specifying using the device letter.

Return Value

Result is a 6 element array. Each element is defined as follows:

Index	Description
0	Model type name for negative polarity device. E.g. 'npn', 'nmos' etc.
1	Model type name for positive polarity device E.g. 'pnp', 'pmos' etc. Empty if device has only a single polarity
2	Device letter. E.g. 'Q' for a BJT
3	Maximum number of terminals.
4	Minimum number of terminals. This is usually the same as the maximum number of terminals, except for BJTs whose substrate terminal is optional.
5	Value required for LEVEL parameter. 0 means that this is the default device when no LEVEL parameter is specified. -1 will be returned if the 'letter' option is specified.
6	Semi-colon delimited list of valid .MODEL control model name values. E.g. ‘npn’, ‘pnp’ and ‘lpnp’ are returned for the ‘BJT’ device.

GetDeviceParameterName

Type	string	real	string array
Description	device type	level	Options
Compulsory	Yes	No	No
Default		-1	

Return type: string array

Returns string array containing all device parameter names for the specified simulator model type.

Argument 1

Device type specified using its *SPICE letter* e.g. 'Q' for a BJT, 'M' for a MOSFET etc.

Argument 2

Model level if relevant. If omitted or set to -1, the default level for that type of device will be used.

Argu**me**nt 3

String array of length up to 2. May contain one or both of ‘useInternalName’ and ‘readback’. If ‘useInternalName’, then argument 1 must specify the device’s internal name. This is returned by GetInternalDeviceName. Argument 2 is ignored in this case.

If ‘readback’ is specified, the function returns names of ‘read back’ parameters. Read back parameters aren’t writeable but return information about a device’s operating characteristics. For example, most MOS devices have ‘vdsat’ read back parameter that returns the saturation voltage. This function only returns the names of read back parameters. To find their values, use GetInstanceParamValues.

Return value

String array of length determined by the number of parameters the device has. Each element contains the name of a single parameter. To find the values for the parameters use GetInstanceParamValues.

Example

The following:

Show GetDeviceParameterNames('M')

returns:

0	'L'
1	'W'
2	'M'
3	'AD'
4	'AS'
5	'PD'
6	'PS'
7	'NRD'
8	'NRS'
9	'IC-VDS'
10	'IC-VGS'
11	'IC-VBS'
12	'TEMP'

GetDriveType

Type	string
Description	path
Compulsory	Yes
Default	

Return type: string

Determines the type of drive or file system of the specified path. Returns one of the following values:

Return value	Description
‘local’	Drive or file system present on the local machine
‘remote’	Network drive or file system
‘cdrom’	CD Rom or DVD drive
‘other’	Other file system or drive
‘notexist’	The path doesn’t exist or media not present.
‘unknown’	Drive type or file system could not be determined

GetEnvVar

Arguments:

Type:	string
Description:	system environment variable name
Compulsory:	Yes
Default:	

Return type: string

Returns the value of a system environment variable.

GetFile

Arguments:

Type:	string	real
Description:	File specification	0: file must exist, 1:need not exist
Compulsory:	Yes	No
Default:		0

Return type: string

Opens "Open File" dialog box. Return value is full pathname of file selected by user. If user cancels operation, function returns an empty string. Argument to function supplies description of files and default extension. These two items are separated by '\'. E.g. getfile('Schematic Files\sch') .

This function has now been superseded by the function **GetUserFile** which is more flexible.

GetFileCd

Arguments:

Type:	string	real
Description:	File specification	0: file must exist, 1:need not exist
Compulsory:	Yes	No
Default:		0

Return type: string

Opens "Open File" dialog box. Return value is full pathname of file selected by user. Current directory is changed to directory holding selected file. If user cancels operation, function returns an empty string. Argument to function supplies description of files and default extension. These two items are separated by '\'. E.g. getfilecd('Text Files\txt') .

This function has now been superseded by the function **GetUserFile** which is more flexible.

GetFileExtensions

Arguments:

Type:	string
Description:	file type
Compulsory:	Yes
Default:	

Return type: string array

Returns a string array containing all valid extensions (without prefixed '.') for the given file type. The extension returned in the first element is the default. File extensions can be changed in the general options dialog box (**File|Options|General...**) and are stored in a number of option variables. These are listed in the following table.

GetFileExtensions argument	Used for	Option name	Default
'Data	Data files	DataExtension	sxdat, dat
'Text'	Text files	TextExtension	txt, net, cir, mod, ldf, sxscr, lib, lb, cat
'LogicDef'	Logic definition files used with arbitrary logic block	LogicDefExtension	ldf
'Script'	Script files	ScriptExtension	sxscr, txt
'Model'	Model files	ModelExtension	lb, lib, mod, cir
'Catalog'	Catalog files	CatalogExtension	cat

GetFileSave

Arguments:

Arguments:	
Type:	string
Description:	file specification
Compulsory:	Yes
Default:	

Return type: string

Opens "Save File" dialog box. Return value is full pathname of file selected by user. If user cancels operation, function returns an empty string. Argument to function supplies description of files and default extension. These two items are separated by '\'. E.g. getfile('Text Files\txt') . User will be warned if an existing file is selected.

This function has now been superseded by the function **GetUserFile** which is more flexible.

GetFonts

No arguments

Return type: string array

Returns the names of all objects in the program whose font may be edited. The function is usually used in conjunction the function GetFontSpec, the function SelectFontDialog and the command EditFont.

GetFontSpec

Type	string
Description	Object name
Compulsory	Yes
Default	

Return type: string

Returns the current font specification for the object whose name is passed to argument 1. Valid object names can be obtained from the GetFonts function. The return value may be used to initialise the SelectFontDialog which allows the user to define a new font.

The return value represents the font of the object as a string consisting of a number of values separated by semi-colons. The values define the font in terms of its type face, size, style and other characteristics. However, these values should not be used directly as the format of the string may change in future versions of the product. The return value should be used only as an argument to functions or commands that accept a font definition. E.g. The SelectFontDialog function and EditFont command.

If the object name passed is not recognised the function will return the definition for the default font.

GetGraphObjects

Type	string	string
Description	Object type name	Graph id
Compulsory	No	No
Default		

Return type: string array

Returns a list of IDs for the graph objects defined by the optional arguments as follows:

If no arguments are specified, the IDs for all graph objects are returned.

If the first argument is specified, all objects of the defined type will be returned

If both arguments are specified, all objects of the defined type and located on the specified graph will be returned.

If the type name is invalid, or if the graph id specified in arg 2 is invalid or if there are no graphs open, the function will return an empty vector.

GetGraphObjPropNames

Type	string
Description	Graph object ID
Compulsory	Yes
Default	

Return type: string array

Returns the valid property names for the graph object defined by argument 1

GetGraphObjPropValue

Type	string	string
Description	Graph object ID	Property name
Compulsory	Yes	No
Default		Return all values

Return type: string array

Returns property values for the specified object. If argument 2 is present the value of one particular property will be returned. Otherwise the function will return an array containing

all property values. The order of the values corresponds to the return value of GetGraphObjPropNames.

(Note the function GetGraphObjPropValues is the same but will only accept one argument)

GetGraphTitle

No Arguments

Return type: string

Returns title of currently selected graph.

GetGroupInfo

Type	string
Description	group name
Compulsory	Yes
Default	

Return type: string array

Returns information about a group.

Argument 1

Group name for which information is required. Enter ‘’ to obtain information on the current group.

Return Value

String array of length 3 as described in the following table:

Index	Description
0	Source file. This is the path name for the file that contains the data for the group. If the groups data is stored in RAM, this element will hold an empty string
1	Group title. For groups created by a simulation (which is to say virtually all groups) this is obtained from the netlist title
2	Empty - reserved for future use

GetGroupStepParameter

Type	string
Description	Group name
Compulsory	No
Default	Current group

Return type: string

Returns the name of the ‘stepped parameter’ of a multi-step run. This value is stored within the group created for the simulation run's output data. The stepped parameter is a label that identifies the parameter, device, model parameter or other quantity that is varied during a multi-step run.

GetGroupStepVals

Type	string
Description	Group name
Compulsory	No
Default	Current group

Return type: real array

Returns the ‘stepped values’ in a multi-step run. These values are stored within the group created for the simulation run's output data. The stepped values are the values assigned to the ‘stepped parameter’during a multi-step run.

GetInstanceParamValues

Type	string	string	string
Description	Instance name	Parameter name	Options
Compulsory	Yes	No	No
Default		Get all parameters	

Return type: string or string array

Returns simulation instance parameter values for the device specified. This function returns the values used in the most recent simulation. If simulation has been run, or it was aborted or reset (using Reset command), then this function will return an empty vector.

If argument 3 is set to ‘readback’, this function will return the values for readback parameters.

Argument 1

Instance name, e.g. Q23, R3 etc. This is the name used in the netlist stripped of its dollar prefix if applicable.

Argument 2

Name of parameter whose value is required. If this argument is missing or empty, then all parameters will be returned. The number and order of the parameters in this case will match the return value of parameter names from the function GetDeviceParameterNames.

Argument 3

If set to ‘readback’ and argument 2 is empty, this function will return the values of all read back values for the devices. ‘read back’ values are values calculated during a run and give useful information about a device’s operating conditions. Note that the value returned will reflect the state of the device at the last simulation point. For example, if a transient run has just been performed, the values at the final time point will be given. If a small-signal analysis has been performed, the results will usually reflect the DC operating point conditions.

Return Value

If argument 2 is provided and valid, will return a single string expressing the value of the parameter. If arg 2 is missing or empty, a string array will be returned with all parameter values.

GetInternalDeviceName

Type	string array
Description	Model details
Compulsory	Yes
Default	

Return type: string

Finds the simulator's internal device name for a model defined using its model type name and optionally, level and version.

The internal device name is a unique name used to define a primitive simulator device. For example, npn and pnp transistors have the internal device name of 'BJT'.

Level 1 MOSFETs have the internal device name of 'MOS1' while nmos level 8 devices are called 'BSIM3'. Some functions - e.g. GetDeviceInfo - require the internal device name as an argument.

Argument 1

1 - 3 element string array which describes device.

Index	Description
0	Model type name as used in the .MODEL control. E.g. 'nmos', 'npn' etc.
1	Optional. Value of LEVEL parameter. If omitted, default level is assumed
2	Optional. Value of VERSION parameter.

GetLastError

No Arguments

Return type: string

Returns a string with one of three values signifying the status of the most recent command executed. The three values are:

'OK'	Command executed without error
'Error'	One or more errors occurred in the most recent command
'Fatal'	The most recent command was not recognised or the evaluation of a braced substitution failed.

The command switches /noerr and /quiet (see **Command Line Switches** in the Pulsonix Spice User's manual) can be used to effectively disable non-fatal errors. This function allows customised action in the event of an error occurring. For example, if a simulation fails to converge, the run command yields an error. This function can be used to take appropriate action in these circumstances.

When a fatal error occurs, the command will abort unconditionally and this function returns 'Fatal'.

GetLegendProperties

Arguments:

Type:	string	string
Description:	curve id	options
Compulsory:	Yes	No
Default:		'names'

Return type: string array

Returns either all legend property names or all legend property values for specified curves. Legend Properties are the text associated with curve names in the graphs "Legend Panel".

If argument 2 = 'values' the function returns legend property values. Otherwise it returns legend property names.

GetMenuItems

Type	string
Description	Menu path
Compulsory	Yes
Default	

Return type: string array

Returns all menu item names in the specified menu.

Argument 1

Specifies the path for the menu as it would be provided to the DefMenu command but without the menu item name.

Return Value

Returns a string array listing all the menu item names. E.g.

GetMenuItems('Shell|&File')

returns all the menu items in the command shell's File menu.

GetModelFiles

No arguments

Return type: string array

Returns a list of currently installed device models.

GetModelName

Type	string
Description	Instance name
Compulsory	Yes
Default	

Return type: string

Returns the model name used by an instance. The model name is the name for the parameter set (e.g. 'QN2222') as opposed to 'model type name' (e.g. 'npn') and 'internal device name' (e.g. 'BJT').

Note that all simulator devices use a model even if it is not possible for the device to use a .MODEL statement. Inductors, for example. are not permitted a .MODEL control but they nevertheless all refer to an internal model which is always called '$Inductor'.

GetModelParameterNames

Type	string
Description	Internal device name
Compulsory	Yes
Default	

Return type: string array

Returns the names or default values of all real valued parameters for a device model.

Argument 1

Internal device name. This is returned by GetInternalDeviceName GetModelType

Argument 2 - UNSUPPORTED

If a second argument is supplied set to 'default', the function will instead return the default values used for the device's parameter names. This doesn't work correctly for all simulator devices and so is currently unsupported.

GetModelParameterValues

Type	string	string
Description	Model name	Parameter name
Compulsory	Yes	No
Default		All values returned if omitted

Return type: string array

Returns the values of all parameters of the specified model. (Defined by ‘model name’ e.g. 'Q2N2222'). This function reads the values from the simulator and requires that a simulation has been run or checked. The returned array with arg2 omitted is of the same size as the array returned by GetModelParameterNames for the same device and the values and parameter names map directly.

Argument 1

Model name. (Model name is the user name for a model parameter set as defined in the .MODEL control e.g. 'Q2N2222').

Argument 2

Parameter name. If specified return value will be a single value for the specified parameter. If omitted, the values for all parameters will be returned.

GetModelType

Type	string
Description	Model name
Compulsory	Yes
Default	

Return type: string

Returns internal device name given user model name. The internal device name is a name used internally by the simulator and is required by some functions. GetInternalDeviceName. The user model name is the name of a model parameter set defined using .MODEL. E.g. 'Q2N2222'.

Important: this function only works for models used by the current simulation. That is, you must run or check a simulation on a netlist that uses the specified model before calling this function.

Argument 1

User model name. See above.

GetNonDefaultOptions

No arguments

Return type: string array

Returns names of all .OPTION settings in the most recent simulation that were not at their default value.

GetNumCurves

Arguments:

Type:	string
Description:	option name
Compulsory:	Yes
Default:	

Return type: real

Returns the number of curves in curve group. This is applicable to curves plotted for a Monte Carlo analysis.

GetOption

Arguments:

Type:	string
Description:	option name
Compulsory:	Yes
Default:	

Return type: string

Returns the value of the "Option" of name given as argument. Options are created using the **Set** command - see the Pulsonix Spice User's manual for details on Simulation Options. The **GetOption** function returns 'FALSE' if the option does not exist and 'TRUE' if it exists but has no value.

GetPath

Arguments:

Type:	string
Description:	Item name
Compulsory:	Yes
Default:	

Return type: real

Returns full path name of one of the following.

Argument value	Function
ScriptDir	Script directory
StartUpDir	Start up directory
StartUpFile	Start up script
BiScriptDir	Built-in script directory
ExeDir	Directory containing executable file.
TempDataDir	Temporary simulation data directory
DocsDir	File system directory for the "My Documents" folder

GetPlatformFeatures

No arguments

Return type: string array

Returns information on availability of certain features that are platform dependent.

Return value

Currently a string of length 4 defined as follows

Index	Description
0	Is ‘ShellExecute’ function implemented. ‘true’ or ‘false’
1	Obsolete
2	Is ‘VersionStamp’ function implemented. ‘true’ or ‘false’
3	Is context sensitive help implemented. ‘true’ or ‘false’

GetPrinterInfo

No arguments

Return type: string array

Returns array of strings providing system printer names and current application default printer. The return value for index 0 will always be -1. Format is as follows:

Index	Description
0	Number of printers available in system
1	Index of printer that is currently set as default. (This is the default for the application *not* the system default printer - see below)
2 onwards	List of printer names.

Example

The following is an example of executing the command “Show GetPrinterInfo”

Index	GetPrinterInfo()
0	'4'
1	'2'
2	'HP LaserJet 4L to file'
3	'HP LaserJet 4L'
4	'Acrobat PDFWriter'
5	'Acrobat Distiller'

The default index is 2 so this means that 'Acrobat PDFWriter' is currently set as the default printer. This is the current default for the *application* and is what will be set when you open a Print dialog box. When Pulsonix starts, it will be initialised to the *system* default printer but changes whenever you select a different printer in any of the printer dialogs.

GetPrintValues

No arguments

Return type: string array

Returns the names of all quantities specified in .PRINT controls in the most recent simulation run.

GetSelectedCurves

No Arguments

Return type: string array

Returns array of curve id's for selected curves.

GetSelectedGraphAnno

No arguments

Return type: string

Returns the ID for the currently selected graph annotation object. If no object is selected, the function returns '-1'. If no graphs are open, the function returns an empty vector.

See”Graph Objects” for information on graph annotation objects.

GetSelectedYAxis

No Arguments

Return type: string

Returns id of selected y-axis.

GetSimConfigLoc

No arguments

Return type:string

Returns the location of the simulator's configuration information. This function returns its result in an identical form to the GetConfigLoc function.

GetSimulationInfo

No arguments

Return type:string array

Returns a string array of length 11 providing the following information about the most recent simulation:

Index	Description
0	Netlist path
1	List file path
2	Using data file 'True' or 'False'
3	Name of user specified data file
4	Collection name (obsolete)
5	.OPTIONS line specified at RUN command
6	Analysis line specified at RUN command
7	Reserved for future use
8	Reserved for future use
9	Reserved for future use
10	Reserved for future use

GetSimulationSeeds

No arguments

Return type:real array

Returns the seeds used for the most recent run. If this run was a Monte Carlo analysis, the return value will be an array of length equal to the number of Monte Carlo steps. Each element will hold the seed used for the corresponding step.

GetSimulatorOption

Type	string
Description	Option name
Compulsory	Yes
Default	

Return type: string

Returns the value of a simulator option as used by the most recent analysis. The argument may be any one of the option names defined for the .OPTIONS control. E.g.

GetSimulatorOption('RELTOL')

will return the value of RELTOL for the most recent run. If the option value was not explicitly specified in a .OPTIONS control, its default value will be returned.

GetSimulatorStats

No arguments

Return type: real array

Returns a 26 element real array providing statistical information about the most recent run. The meaning of each field is described below:

Index	Description
0	Number of event driven outputs
1	Number of event driven ports
2	Number of event driven instances
3	Number of event driven nodes
4	Number of equations (= matrix dimension = total number of nodes including internal nodes)
5	Total number of iterations
6	Number of transient iterations
7	Number of JI2 iterations. (First attempt at DC bias point)
8	Number of GMIN iterations
9	Number of source stepping iterations
10	Number of pseudo transient analysis iterations
11	Number of time points
12	Number of accepted time points
13	Total analysis time
14	Transient analysis time
15	Matrix load time. (The time needed to calculate the device equations)
16	Matrix reorder time.
17	Matrix decomposition time.
18	Matrix solve time
19	Size of state vector
20	Parameter evaluation time
21	Matrix decomposition time (transient only)
22	Matrix solve time (transient only)
23	Circuit temperature
24	Circuit nominal temperature
25	Number of matrix fill-ins
26	Simulator Initialisation time
27	Number of junction GMIN iterations
28	Time to process digital events
29	“Accept” time. This is the time used for processing transient time points after the simulator has accepted it. This includes the time taken to write out the data.

GetSimulatorStatus

No arguments

Return type: string

Returns the current status of the simulator. May be one of the following values:

Paused	Simulator paused
InProgress	Simulation in progress. (In practice this cannot happen because there is no method of calling this function while a simulation is in progress)
ConvergenceFail	Last simulation failed because of no convergence
SimErrors	Last simulation failed because of a run time error
NetlistErrors	Last simulation failed because of a netlist error
Warnings	Last simulation completed with warnings

Complete	Last simulation successful
None	No simulation has been run

GetSoaResults

No arguments

Return type: string

Returns the SOA (Safe Operating Area) results for the most recent simulation.

Return Value

Returns an array of strings, each one describing a single SOA failure. Each string is a semi-colon delimited list with fields defined below.

Field	Description
0	SOA Label
1	Start of failure
2	End of failure
3	under' or 'over'. Defines whether the test fell below a minimum limit or exceeded a maximum limit.
4	Value of limit that was violated

GetSystemInfo

No arguments

Return type: string array

Returns information about the user's system

Return Value

String array of length 7 as defined by the following table:

Index	Description
0	Computer name
1	User log in name
2	Returns 'Admin' if logged in with administrator privilege otherwise returns 'User'.
3	Available system RAM in bytes
4	Operating system class. Returns 'WINNT'. With earlier versions this maybe WIN9X if running on Windows 95, 98 or ME. These platforms are not supported by version 2.
5	Operating System description. Returns descriptive name for operating system.

GetUserFile

Type	string	string	string array	string
Description	file filter	default extension	options	initial file
Compulsory	No	No	No	No
Default			<<empty>>	<<empty>>

Return type: string

Function opens a dialog box to allow the user to select a file.

Argument 1

Defines file filters. The ‘save as type’ list box may contain any number of entries that defines the type of file to be displayed. This argument defines the entries in this list box.

Each entry consists of a description followed by a pipe symbol ('|') then a list of file extensions separated by semi-colons (';'). Entries are also separated by the pipe ('|") symbol. For example: to list just data files enter:

“Data files|*.sxdat;*.dat'”

Note that the text is enclosed in both single and double quotes. Strings in expressions are denoted by single quotes as usual but the semi-colon is normally used to separate commands on a single line. This is inhibited by enclosing the whole string in double quotes.

If you wanted to provide entries for selecting - say - both data and netlists, you could use the following:

“Data files|*.sxdat;*.dat|Netlist files|*.net;*.cir'”

Argument 2

The default extension specified without the dot. This is the extension that will automatically be added to the file name if it does not already have one of the extensions specified in the filter.

Argument 3

String array that specifies a number of options. Any or all of the following may be included:

'ChangeDir'	If present, the current working directory will change to that containing the file selected by the user
'Open'	If present a File Open box will be displayed other wise a Save As box will be displayed.
'NotExist'	If used with 'Open', the file is not required to already exist to be accepted
'ShowReadOnly'	If present and 'Open' is also specified, an Open as read- only check box will be displayed. The user selection of this check box will be returned in either the second or third field of the return value.
'FilterIndex'	If specified, the type of file selected by the user will be returned as an index into the list of file filters specified in argument 1. So, 0 for the first, 1 for the second etc.

Argument 4

Initial file selection.

Return value

String array of length between 1 and 3 as described in the following table:

Option 'ShowReadOnly'	Option 'FilterIndex'	Return value
No	No	Path name only
Yes	No	2 element array: index=0 path name index=1 Read only checked - 'TRUE' or 'FALSE'
No	Yes	2 element array index=0 path name index=1 Filter index selected
Yes	Yes	3 element array index=0 path name index=1 Filter index selected index=2 Read only checked - 'TRUE' or 'FALSE'

GetVecStepParameter

Type	real/complex array
Description	vector
Compulsory	Yes
Default	

Return type: string

This function retrieves the name of the parameter that was stepped to obtain the vector data supplied. It will only return a meaningful result for data vectors generated by a multi-step analysis. For example, if an analysis was performed which stepped the value of the resistor R7, this function would return 'R7' when applied to any of the data vectors created by the simulator. If the analysis was a Monte Carlo run, the function will return 'Run'.

If this function is applied to single division data as returned by a normal single step run, the return value will be an empty vector.

GetVecStepVals

Type	real/complex array
Description	vector
Compulsory	Yes
Default	

Return type: real array

This function retrieves the values assigned to the parameter that was stepped to obtain the vector data supplied. It will only return a meaningful result for data vectors generated by a multi-step analysis. For example, if an analysis was performed which stepped the value of the resistor R7 from 100Ω to 500Ω in 100Ω steps, this function would return [100, 200, 300, 400, 500]. If the analysis was a Monte Carlo run, the function will return the run numbers starting from 1.

If this function is applied to single division data as returned by a normal single step run, the return value will be an empty vector.

GetWindowNames

No Arguments

Return type: string array

Returns names of current windows. Result can be supplied as argument to Focus command using /named switch.

GetXAxis

No Arguments

Return type: string

Returns the id of the x-axis in the currently selected graph.

GraphLimits

No Arguments

Return type: real array

The x and y axis limits of the currently selected graph and axis type (log/linear). Function will fail if there are no selected graphs. Meaning of each index of the 6 element array are as follows:

0	x-axis lower limit
1	x-axis upper limit
2	y-axis lower limit
3	y-axis upper limit
4	1 if x-axis is logarithmic, 0 if linear
5	1 if y-axis is logarithmic, 0 if linear

GroupDelay

Arguments:

Type:	real/complex array
Description:	vector
Compulsory:	Yes
Default:	

Return array: real array

Returns the group delay of the argument. Group delay is defined as:

$$\frac{d(phase(y))}{dx} \cdot \frac{1}{2.\pi}$$

where y is the supplied vector and x is its reference. The **GroupDelay** function expects the result of AC analysis where y is a voltage or current and its reference is frequency.

This function will yield an error if its argument is complex and has no reference.

Groups

Arguments:

Type:	string
Description:	Title|Name
Compulsory:	No
Default:	'name'

Return type: string array

Returns names of available groups. The first element (with index 0) is the current group. If the argument 'Title' is provided, the full title of the group is returned. More information about groups can be found in Accessing Simulation Data.

GroupsInCollection

Arguments

Type:	string
Description:	Collection name
Compulsory:	Yes
Default:	

Return type: string array

Returns array of group names belonging to the specified named collection. For information on collections see Accessing Simulation Data.

HasLogSpacing

Type	real
Description	Vector
Compulsory	Yes
Default	

Return type: real

Performs a simple test to determine whether the supplied vector is logarithmically spaced. The return value is 1.0 if the vector is logarithmically spaced and 0.0 otherwise. Note the function expects to be supplied with x-values.

Histogram

Arguments

Type:	real array	real
Description:	Vector	Number of bins
Compulsory:	Yes	Yes
Default:		

Return type: real array

Creates a histogram of argument 1 with the number of bins specified by argument 2. The bins are divided evenly between the maximum and minimum values in the argument.

Histograms are useful for finding information about waveforms that are difficult to determine by other means. They are particularly useful for finding "flat" areas such as the flat tops of pulses as these appear as well defined peaks. The Histogram() function is used in the rise and fall time scripts for this purpose.

User's should note that using this function applied to raw transient analysis data will produce misleading results as the values are unevenly spaced. If you apply this function to simulation data, you must either specify that the simulator outputs at fixed intervals (select the **Output at interval** option in the Pulsonix Schematics **Simulator Parameters...** dialog box) or you must interpolate the results using the **Interp** function.

Iff

Arguments:

Type:	real array	real array	real array
Description:	Vector to be filtered	Coefficients	Initial conditions
Compulsory:	Yes	Yes	No
Default:			zero

Return type: same as args 2 and 3

If the first argument evaluates to TRUE (i.e. non-zero) the function will return the value of argument 2. Otherwise it will return the value of argument 3. Note that the type of arguments 2 and 3 must both be the same. No implicit type conversion will be performed on these arguments.

IIR

Arguments:

Type:	real array	real array	real array
Description:	Vector to be filtered	Coefficients	Initial conditions
Compulsory:	Yes	Yes	No
Default:			zero

Return type: real array

Performs "Infinite Impulse Response" digital filtering on supplied vector. This function performs the operation:

$y_n = x_n \cdot c_0 + y_{n-1} \cdot c_1 + y_{n-2} \cdot c_2 \ldots$

Where:

x is the input vector (argument 1)

c is the coefficient vector (argument 2)

y is the result (returned value)

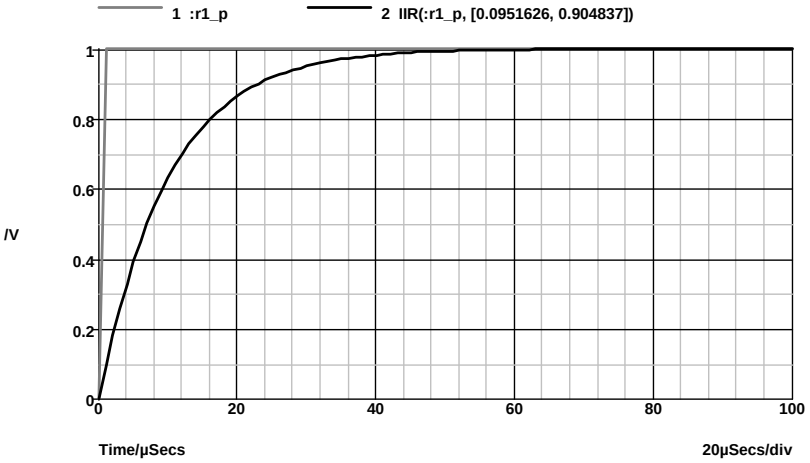
The third argument provide the "history" of y i.e. y_{-1}, y_{-2} etc. as required.

The operation of this function (and also the **FIR** function) is simple but its application can be the subject of several volumes! In principle an almost unlimited range of IIR filtering operations may be performed using this function. Any text on Digital Signal Processing will provide further details.

User's should note that using this function applied to raw transient analysis data will not produce meaningful results as the values are unevenly spaced. If you apply this function to simulation data, you must either specify that the simulator outputs at fixed intervals (select the **Output at interval** option in the Pulsonix Schematics **Simulator Parameters...** dialog box) or you must interpolate the results using the **Interp** function.

Example

The following graph shows the result of applying a simple first order IIR filter to a step:



The coefficients used give a time constant of 10 * the sample interval. In the above the sample interval was 1µSec so giving a 10µSec time constant. As can be seen a first order IIR filter has exactly the same response as an single pole RC network. A general first order function is:

$y_n = x_n \cdot c_0 + y_{n-1} \cdot c_1$

where $c_0=1-\exp(-T/\tau)$

and $c_1=\exp(-T/\tau)$

and τ=time constant

and T=sample interval

The above example is simple but it is possible to construct much more complex filters using this function. While it is also possible to place analog representations on the circuit being simulated, use of the IIR function permits viewing of filtered waveforms after a simulation run has completed. This is especially useful if the run took a long time to complete.

im

Arguments

Type:	real/complex array
Description:	
Compulsory:	Yes
Default:	

Return type: real array

Returns imaginary part of argument.

imag

Identical to **im()**

InputGraph

Arguments:

Type:	string	string
Description:	initial text	message
Compulsory:	No	No
Default:		<<empty>>

Return type: string

Opens a simple dialog box prompting the user for input. Dialog box position is chosen to keep selected graph visible if possible. Argument provides initial text, return value is text entered by user. The function returns an empty vector if the user cancels the dialog box.

Integ

Arguments:

Type:	real array
Description:	vector
Compulsory:	Yes
Default:	

Return type: real array

Integrates the argument with respect to its reference. See “Vector References” for details.

The function uses simple trapezoidal integration.

An error will occur if the argument supplied has no reference.

Interp

Arguments:

Type:	real array	real	real	real
Description:	Vector to be interpolated	Number of points	Interpolation order	include last point
Compulsory:	Yes	Yes	No	No
Default:			2	FALSE

Returns a vector with length specified by argument 2 obtained by interpolating the vector supplied as argument 1 at evenly spaced intervals. The optional third argument specifies the interpolation order. This can be any integer 1 or greater but in practice there are seldom reasons to use values greater than 4. If argument 4 is TRUE (1) the final point of the interpolated result will coincide with the final point of the input vector and the interval

between points is T/(N-1) where T is the interval of the whole input vector and N is the number of points. If argument 4 is FALSE (0) the interval is T/N and the final point is at a location T/N before the final input point. The latter behaviour is compatible with earlier versions and is also what should be used if the function is interpolating data to be used by the FFT function.

The Interp() function overcomes some of the problems caused by the fact that raw transient analysis results are unevenly spaced. It is used by the FFT plotting scripts to provide evenly spaced sample points for the FFT() function.

IsComplex

Arguments:

Type:	any
Description:	vector
Compulsory:	Yes
Default:	

Return type: real

Returns 1 (=TRUE) if the supplied argument is complex and 0 (=FALSE) if the argument is any other type

IsFullPath

Type	string
Description	path
Compulsory	Yes
Default	

Return type: real

Returns TRUE if the supplied path name is a full absolute path.

Argument 1

File system path name

Return Value

TRUE if arg is a full absolute path. FALSE if it is a relative path.

IsModelFile

Type	string	string
Description	Path of file	Option
Compulsory	Yes	No
Default		

Return type: real

Returns 1 if the specified file contains .MODEL, .SUBCKT or .ALIAS definitions. Otherwise returns 0. The function will unconditionally return 0 if the file has any of the following extensions:

.EXE, .COM, .BAT, .PIF, .CMD, .SCH, .SXSCH, .SXDAT,.SXGPH

This will be overridden if the second argument is set to 'AllExt'.

IsNum

Arguments:

Type:	any
Description:	vector
Compulsory:	Yes
Default:	

Return type: real

Returns 1 (=TRUE) if the supplied argument is numeric (real or complex) and 0 (=FALSE) if the argument is a string

IsScript

Type	string
Description	script name
Compulsory	Yes
Default	

Return type: real

Function to determine whether the supplied script name can be located. Calling this script will fail if this function returns FALSE. Note that the function doesn't check the script itself. It only determines whether or not it exists.

Argument 1

Script name

Return Value

Returns TRUE if the supplied script name can be located in the standard script path.

IsStr

Arguments:

Type:	any
Description:	vector
Compulsory:	Yes
Default:	

Return type: real

Returns 1 (=TRUE) if the supplied argument is a string and 0 (=FALSE) if the argument is numeric (real or complex).

Length

Arguments:

Type:	any
Description:	vector
Compulsory:	Yes
Default:	

Return type: real

Returns the number of elements in the argument. The result will be 1 for a scalar and 0 for an empty value.

The **Length** function is the only function which will not return an error if supplied with an "empty" value. Empty variables are returned by some functions when they cannot produce a return value. All other functions and operators will yield an error if presented with an empty value and abort any script that called it.

ListDirectory

Type	string	string
Description	Path specification	Option
Compulsory	Yes	No
Default		'none'

Return type: string array

Lists all files that comply with the spec provided in argument 1.

Argument 1

Specification for output. This would usually contain a DOS style wild card value. No output will result if just a directory name is given.

Argument 2

If omitted, the result will be file names only. If set to 'fullpath', the full path of the files will be returned.

ln

Arguments:

Type:	real/complex array
Description:	vector
Compulsory:	Yes
Default:	

Return type: real/complex array

Returns the natural logarithm of the argument. If the argument is real and 0 or negative an error will result. If the argument is complex it will return a complex result even if the imaginary part is 0 and the real part negative. An error will always occur if both real and imaginary parts are zero.

Locate

Arguments:

Type:	real	real
Description:	vector	search value
Compulsory:	Yes	Yes
Default:		

Return type: real

Function performs a binary search on the input vector (argument 1) for the value specified in argument 2. The input vector must be monotonic i.e. either always increasing or always reducing. This is always the case for the reference vector see "Vector References" of a simulation result. If the input vector is increasing (positive slope) the return value is the index of the value immediately *below* the search value. If the input vector is decreasing (negative slope) the return value is the index of the value immediately *above* the search value.

log

Identical to **ln**.

log10

Arguments:

Type:	real/complex array
Description:	vector
Compulsory:	Yes
Default:	

Return type: real/complex array

Returns log to base 10 of argument. If the argument is real and 0 or negative an error will result. If the argument is complex it will return a complex result even if the imaginary part is 0 and the real part negative. An error will always occur if both real and imaginary parts are zero.

mag

Arguments:

Type:	real/complex array
Description:	vector
Compulsory:	Yes
Default:	

Return type: real array

Returns the magnitude of the argument. This function is identical to the **abs** function.

magnitude

Identical to mag

MakeCollection

Type	string	string
Description	base name	Obsolete
Compulsory	Yes	No
Default		

Return type: string

Creates a collection and returns its assigned name. Collections are used to combine groups that are related to each other. See “Collections” for details about collections.

The first argument is the base name. The actual name used (which is returned) is the base name appended with a number assigned by the function to make the name unique.

Collections were originally developed for earlier versions of Pulsonix and were used to handle data from multi-step analyses such as Monte Carlo. They are no longer used by the front end but the functions and commands that access them are still available for user applications.

MakeDir

Arguments:

Type:	string
Description:	Directory name
Compulsory:	Yes
Default:	

Return type: real

MakeString

Type	real	string array
Description	Number of elements in result	Initial values
Compulsory	Yes	No
Default		

Return type: string array

Creates an array of strings. Length of array is given as argument to function. The strings may be initialised by supplying argument 2.

Argument 1

Number of elements to create in string array.

Argument 2

Initialises values of string. Can be used to extend an existing string. e.g:

Let str = ['john', 'fred', 'bill']

```
Let str = MakeString(6, str)
```

In the above the string `str` will be extended from length 3 to length 6 by the call to `MakeString`.

Return Value

Returns new string

Max

Type	real	real
Description	vector 1	vector 2
Compulsory	Yes	Yes
Default		

Return type: real array

Returns an array equal to the length of each argument. Each element in the array holds the larger of the corresponding elements of argument 1 and arguments 2

Maxidx

Arguments:

Type:	real/complex array
Description:	vector
Compulsory:	Yes
Default:	

Return type: real

Returns index of the array element in argument 1 with the largest magnitude.

Maxima

Type	real array	real array	string
Description	vector	[min limit, max limit]	options
Compulsory	Yes	No	No
Default		[-□, +□]	<<empty>>

Return type: real array

Returns array of values holding every maximum point in the supplied vector whose value complies with limits specified in argument 2.

Argument 1

Input vector

Argument 2

Real array of max length 2. Specifies limits within which the input values must lie to be included in the result.

0	Minimum limit i.e. maxima must be above this to be accepted
1	Maximum limit i.e. maxima must be below this to be accepted.

Argument 3

String array of max length 2. Specifies two possible options:

'xsort'	If specified the output is sorted in order of their x-values (reference). Otherwise the values are sorted in descending order of y magnitude.
'nointerp'	If *not* specified the values returned are obtained by fitting a parabola to the maximum and each point either side then calculating the x, y location of the point with zero slope. Otherwise no interpolation is carried out and the literal maximum values are returned.
'noendpts'	If specified, the first and last points in the data will not be returned as maximum points.

Return value

The function returns the XY values for each maximum point. The X-values are returned as the vector's reference.

Maximum

Type	real/complex array	real	real
Description	Vector	Min range	Max range
Compulsory	Yes	No	No
Default		start of vector	end of vector

Return type: Real

Returns the largest value found in the vector specified in argument 1 in the range of x values specified by arguments 2 and 3

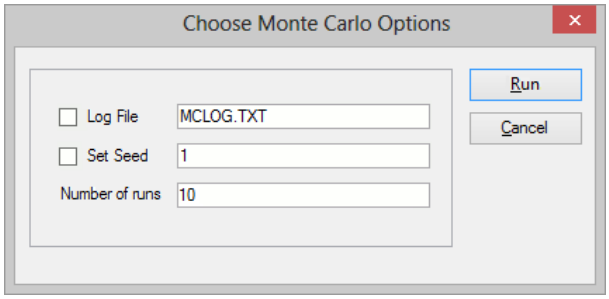
MCOptions

Arguments:

Type:	real	string	real
Description:	number of runs	log file name	seed value
Compulsory:	No	No	No
Default:	10	empty	-1 (no seed)

Return type: string array

This is a special purpose function designed as part of the Monte Carlo analysis user interface. It opens a dialog box displaying Monte Carlo options:



The three arguments initialise the controls. If no values are supplied for log file and set seed, the associated check boxes will be clear. The return vector gives the values entered by the user. If the "Set Seed" check box is clear a value of -1 is returned for the seed. If the "Log File" check box is clear an empty string is returned for the log file. Note that the array returned contains only strings even though two of the results are actually numbers.

mean

Arguments:

Type:	real/complex array
Description:	vector
Compulsory:	Yes
Default:	

Return type:

Returns the average of all values in supplied argument. If the argument is complex the result will also be complex.

Mean1

Arguments:

Type:	real array	real	real
Description:	Input vector	start x value	end x value
Compulsory:	Yes	No	No
Default:		Start of input vector	End of input vector

Return type: real

Returns the integral of the supplied vector between the ranges specified by arguments 2 and 3 divided by the span (= arg 3 -arg 2). If the values supplied for argument 2 and/or 3 do not lie on sample points, second order interpolation will be used to estimate y values at those points.

MessageBox

Type	string array	string array
Description	Message	Options
Compulsory	Yes	No
Default		

Return type: string

Opens a message dialog box with a choice of styles.

Argument 1

1 or 2 element string array. First element is the text of the message to be displayed in the box. The second element is the box title. If the second element is not supplied the box title will be the name of the application - e.g. ‘Pulsonix Micron AD’

Argument 2

1 or 2 element string array. First element is box style. This may be one of the following:

'AbortRetryIgnore'	Three buttons supplied for user response - Abort, Retry and Ignore
'Ok'	Ok button only
'OkCancel'	Ok and Cancel button
'YesNo'	Yes and No buttons
'YesNoCancel'	Yes, No and Cancel buttons.

Default = 'OkCancel'

Second element is icon style. A small icon is displayed in the box to indicate the nature of the message. Possible values:

'Warn'

'Info'

'Question'

'Stop'

Default = 'Info'

Return value

is a single string indicating the user's response. One of:

'Abort'

'Cancel'

'Ignore'

'No'

'Ok'

'Retry'

'Yes'

Mid

Arguments:

Type:	string	real	real
Description:	String	Start index	Length of result
Compulsory:	Yes	Yes	No
Default:			to end of string

Return type: string

Returns a string constructed from a sub string of argument 1. First character is at index specified by argument 2 while argument 3 is the length of the result. The first character is at index 0.

Example

```
Mid('Hello World!', 6, 5)
```

will return 'World'.

See also

Char

Minidx

Arguments:

Type:	real/complex array
Description:	vector
Compulsory:	Yes
Default:	

Return type: real

Returns index of the array element in argument 1 with the smallest magnitude.

Min

Type	real	real
Description	vector 1	vector 2
Compulsory	Yes	Yes
Default		

Return type: real array

Returns an array equal to the length of each argument. Each element in the array holds the smaller of the corresponding elements of argument 1 and arguments 2

Minima

Type	real array	real array	string
Description	vector	[max limit, min limit]	options
Compulsory	Yes	No	No
Default		[+□, -□]	<<empty>>

Return type: real array

Returns array of values holding every minimum point in the supplied vector whose value complies with limits specified in argument 2.

Argument 1

Input vector

Argument 2

Real array of max length 2. Specifies limits within which the input values must lie to be included in the result.

0	Maximum limit i.e. minima must be below this to be accepted
1	Minimum limit i.e. minima must be above this to be accepted.

Argument 3

String array of max length 2. Specifies two possible options:

'xsort'	If specified the output is sorted in order of their x-values (reference). Otherwise the values are sorted in descending order of y magnitude.
'nointerp'	If *not* specified the values returned are obtained by fitting a parabola to the minimum and each point either side then calculating the x, y location of the point with zero slope. Otherwise no interpolation is carried out and the literal minimum values are returned.
'noendpts'	If specified, the first and last points in the data will not be returned as minimum points.

Return value

The function returns the XY values for each minimum point. The X-values are returned as the vector's reference.

Minimum

Type	real/complex array	real	real
Description	Vector	Min range	Max range
Compulsory	Yes	No	No
Default		start of vector	end of vector

Return type: Real

Returns the smallest value found in the vector specified in argument 1 in the range of x values specified by arguments 2 and 3

ModelLibsChanged

No arguments

Return type: real

Returns 1 if the installed model libraries have been changed since the last call to this function. The function always returns 1 the first time it is called after program start.

norm

Arguments:

Type:	real/complex array
Description:	vector
Compulsory:	Yes
Default:	

Return type: real/complex array

Returns the input vector scaled such that the magnitude of its largest value is unity. If the argument is complex then so will be the return value.

NumDivisions

Type	real/complex array
Description	vector
Compulsory	Yes
Default	

Return type: real

Returns the number of divisions in a vector. Vectors created by multi-step runs such as Monte Carlo are sub-divided into divisions with one division per step. For a full explanation of this concept, see “Multi-division Vectors”.

NumElems

Type	any
Description	vector
Compulsory	Yes
Default	

Return type: real array

Returns the number of elements in a vector. It is similar to the Length function but differs in the way it handles multi-division vectors. NumElems will return an array element for each division in the vector whereas Length will return the number of elements of the first division only.

OpenEchoFile

Type	string	string
Description	File name	Access mode
Compulsory	Yes	Yes
Default		

Return type: real

Redirects the output of the Echo command to a file. Redirection is disabled when the CloseEchoFile function is called or when control returns to the command line.

Argument 1

File name.

Argument 2

A single letter to determine how the file is opened. Can be either ‘w’ or ‘a’. If ‘w’, a new file will be created. If a file of that name already exists, it will be overwritten. If ‘a’ and the file already exists, it will be appended.

Parse

Arguments:

Type:	string	string	string
Description:	Input string	Delimiters	options
Compulsory:	Yes	No	No
Default:		Space, tab	<<empty>>

Return type: string array

Splits up the string supplied as argument 1 into substrings or tokens. The characters specified in argument 2 are treated as separators of the substrings. For example, the following call to Parse():

```
Parse('c:\Spice\work\amp.sch', '\')
```

returns:

```
'c:'
'Spice'
'work'
'amp.sch'
```

If the second argument is omitted, spaces and tab characters will be treated as delimiters. If a space is include in the string of delimiters, tab characters will be automatically added.

If the third arguments is present and equal to 'quoted' the function will treat string enclosed in double quotes as single indivisible tokens.

ParseParameterString

Type	string	string array	string	string
Description	String to parse	Parameter names to process	action	Write value
Compulsory	Yes	Yes	Yes	No
Default				

Return type: string array or scalar

Parses a string of name-value pairs and performs some specified action on them. The function can read specified values and return just the values. It can write to specific values and return a modified string. It can also delete specific values.

Argument 1

String to parse. This is a list of name-value pairs but may also contain any number of unlabelled values at the start of the string. The number of unlabelled values must be specified in argument 3 (see below). Examples:
Without any unlabelled value:

```
W=1u L=2u AD=3e-12 AS=3e-12
```

With 1 unlabelled value

```
2.0 DTEMP=25.0
```

The above shows an equals sign separating names and values, but these may be omitted.

Argument 2

String array listing the names to be processed. If reading (see below) only the values of the names supplied here will be returned. If writing, the names listed in this argument will be edited with new values supplied in argument 4. If deleting, these names will be removed.

Unlabelled parameters may be referenced using the special name '$unlabelled$' followed by the position. I.e. the first unlabelled parameter is position 1, the second 2 and so on. So '$unlabelled$1' refers to the first unlabelled parameter.

Argument 3

1 or 2 element string array. The first element is the action to be performed. The second element is the number of unlabelled parameters that are expected in the input string. This is zero if omitted.

Argument 4

Values to write. These have a 1:1 correspondence with the parameter names in argument 2.

Return Value

If reading, the return value is an array of strings holding the values of the specified parameters. Otherwise it the input string appropriately modified according to the defined action.

Examples

This will return the string array ['1u', '2u']:

```
Let str = 'W=1u L=2u AD=3e-12 AS=3e-12'
```

PathEqual

Type	string array	string array
Description	Path 1	Path 2
Compulsory	Yes	Yes
Default		

Return type: real array

Compares two string arrays and returns a real array of the same length with each element holding the result of a string comparison between corresponding input elements. The string comparison assumes that the input arguments are file system path names and will be case sensitive.

Argument 1

First pathname or pathnames to be compared.

Argument 2

Second pathname or pathnames to be compared.

Return Value

Real array of the same length as the arguments. If the lengths of the arguments are different, an empty vector will be returned. Each element in the array will be either -10, or +1. 0 means the two strings are identical.

ph

Type:	real/complex array
Description:	vector
Compulsory:	Yes
Default:	

Return type: real array

Returns the phase of the argument in degrees.

Each of the function ph(), phase() and phase_rad() produce a continuous output i.e. it does not wrap from 180 degrees to -180 degrees.

This function always returns a result in degrees. This has changed from versions 3.1 and earlier which returned in degrees or radians depending on the setting of the ‘Degrees’ option. For phase in radians, use phase_rad().

phase

identical to **ph**.

phase_rad

Arguments:

Type:	real/complex array
Description:	vector
Compulsory:	Yes
Default:	

Return type: real array

Identical to **ph** and **phase** functions except that the result is always in radians and does not depend on the setting of the option "degrees".

PhysType

Arguments:

Type:	real/complex array
Description:	vector
Compulsory:	Yes
Default:	

Return type: string

Returns the physical type of the argument. Possible values are.

" (meaning dimensionless quantity)

'unknown'

'Voltage'

'Current'

'Time'

'Frequency'

'Resistance'

'Conductance'

'Capacitance'

'Inductance'

'Energy'

'Power'

'Charge'

'Flux'

'Volt^2'

'Volt^2/Hz'

'Volt/rtHz'

'Amp^2'

'Amp^2/Hz'

'Amp/rtHz'

'Volts/sec'

See also

Units

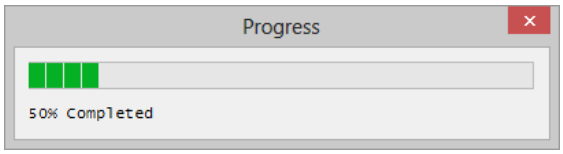
Progress

Arguments:

Type:	real	string array
Description:	Position of progress bar in %	options/control
Compulsory:	Yes	No
Default:		<<empty>>

Return type: real

Opens a dialog box showing a progress bar:



Argument 1

Value from 0 to 100 specifying the position of the bar.

Argument 2

String array of max length 2 used to specify options and control as follows:

'open'	Box is displayed (cannot be used with 'close')
'close'	Box is hidden (cannot be used with 'open')
'smooth'	If specified progress bar is smooth, otherwise it is segmented (as shown above)

Return value

The function returns the value of argument 1

QueryData

Type	string array	string array
Description	Data	Filter
Compulsory	Yes	Yes
Default		

Return type: string array

Filters a list of data items according to search criteria.

Argument 1

The data to be filtered. This should consist of an array of strings comprising semi- colon delimited fields. The filter supplied in argument 2 matches each field to certain criteria and returns the data in the output if those criteria are satisfied.

Argument 2

Filter to determine if data in arg 1 is passed to the output. The filter consists of one or more semi-colon delimited lists which can be combined in Boolean combinations. Each of the lists is compared with the input data for a match and if the resulting Boolean expression is true, the data item is accepted and passed to the return value. Wild cards '*' and '?' may be used in any field. The system is best explained with examples.

Suppose a data item in arg 1 is as follows.

IRFI520N;nmos_sub;X;NMOS;;;;*

and the filter supplied in arg 2 is:

;;X;*;*;*;*;*

This will match successfully. The third fields is the same in both the data and the filter and the remaining filter fields are the '*' wild card which means that anything will be accepted in the corresponding data field.

In the above simple examples, only one filter list has been supplied. However, it is possible to use more sophisticated filters consisting of multiple lists combined using Boolean operators. Boolean operators are specified with the key words:

\OR

\AND

\XOR

\NOT

These can be used to make a Boolean expression using “reverse polish” notation. Here is an example:

['*;nmos;*;*;*;*;*;*', '*;nmos_sub;*;*;*;*;*;*', '\OR']

This will accept any data where the second field is either 'nmos' or 'nmos_sub'. Note that the keyword '\OR' is applied after the filter lists.

As well as the '*' wild card, the '?' may also be used. '?' matches only a single character whereas '*' matches any number of characters. For example:

?mos

Would match 'pmos' as well as 'nmos'. It would also match any other four letter word that ended with the three letters 'mos'.

Return Value

String array of length up to but not exceeding the length of argument 1. Contains all arg 1 items that match the filter as explained above.

RadioSelect

Arguments:

Type:	real	string	string
Description:	Number of button initially selected	Button labels	
Compulsory:	No	No	No
Default:	1	empty	Dialog box caption

Return type: real

Opens a dialog box with up to 6 radio buttons. The number of buttons visible depends on the length of argument 2. All six will be displayed if it is omitted.

The labels for each button is specified by argument 2. The button initially selected is specified by argument 1. Argument 3 provides the text in the dialog boxes caption bar.

The return value identifies the selected button with the top most being 1. If the user cancels the function returns 0.

See also

BoolSelect

EditSelect

ValueDialog

Range

Arguments:

Type:	real/complex array	real	real
Description:	vector	start index	end index
Compulsory:	Yes	No	No
Default:		0	Vector length-1

Return type: real/complex array

Returns a vector which is a range of the input vector in argument 1. The range extends from the indexes specified by arguments 2 and 3. If argument 3 is not supplied the range extends to the end of the input vector. If neither arguments 2 or 3 are supplied, the input vector is returned unmodified.

re

Arguments:

Type:	real/complex array
Description:	vector
Compulsory:	Yes
Default:	

Return type: real array

Returns the real part of the complex argument.

ReadClipboard

No arguments

Return type: string array

Returns text contents of the windows clipboard. Data is returned as one line per array element.

Note that the Show command can be used to write to the clipboard.

ReadConfigSetting

Type	string	string
Description	Section	Key
Compulsory	Yes	Yes
Default		

Return type: string

Reads a configuration setting. Configuration settings are stored in the configuration file. See ‘Configuration Settings’. Settings are defined by a key-value pair and are arranged into sections. The function takes the name of the key and section and returns the value.

Note that option settings (as defined by the Set command) are placed in the ‘Options’ section. Although these values can be read by this function this is not recommended and instead you should always use the GetOption function.

Argument 1

Section name. See description above for explanation.

Argument 2

Key name. See description above for explanation.

Return Value

Value read from configuration file.

See Also

WriteConfigSetting

ReadFile

Arguments:

Type:	string
Description:	File name
Compulsory:	Yes
Default:	

Return type: string array

Returns an array of strings holding lines of text from the file specified by argument 1. The file is expected to contain only ASCII text. The operation will be aborted if non-ASCII characters are encountered.

ReadIniKey

Type	string	string	string
Description	Inifile name	Section name	Key name
Compulsory	Yes	Yes	Yes
Default			

Return type: string array

Reads an INI file. An INI file usually has the extension .INI and is used for storing configuration information. INI files are used by many applications and follow a standard format as follows:

[*section_name1*]

```
key1=value1

key2=value2

...

[section_name2]

key1=value1

key2=value2

...

etc.
```

There may be any number of sections and any number of keys within each section.

The ReadIniKey function can return the value of a single key and it can also return the names of the all the keys in a section as well as the names of all the sections.

Argument 1

File name. You should always supply a full path for this argument. If you supply just a file name, the system will assume that the file is in the WINDOWS directory. This behaviour may be changed in future versions. For maximum compatibility, always use a full path.

Argument 2

Section name. If this argument is an empty string, the function will return the names of the sections in the file.

Argument 3

Key name. If this argument is an empty string and argument 2 is not an empty string, the function will return the names of all the keys in the named section.

ReadRegSetting

Type	string	string	string
Description	Key name	Value name	Top level tree
Compulsory	Yes	Yes	No
Default			'HKCU'

Return type: string

Reads a string setting from the windows registry. Currently this function can only read settings in the HKEY_CURRENT_USER and HKEY_LOCAL_MACHINE top level trees.

Argument 1

Name of key. This must be a full path from the top level. E.g. 'Software\Westdev\Pulsonix\'

Argument 2

Name of value to be read

Argument 3

Top level tree. This may be either 'HKEY_CURRENT_USER' or 'HKEY_LOCAL_MACHINE' or their respective abbreviations 'HKCU' and 'HKLM'.

Return Value

Returns value read from the registry. If the value doesn't exist, the function returns an empty vector.

real

Identical to **re**.

Ref

Arguments:

Type:	real/complex array
Description:	vector
Compulsory:	Yes
Default:	

Return type: real/complet array

Returns the reference of the argument. See "Vector References".

RefName

Arguments:

Type:	real/complex array
Description:	vector
Compulsory:	Yes
Default:	

Return type: string

Returns the name of the reference of the supplied vector. See "Vector References". Note that the **Ref** function (above) returns the actual data for the reference.

RelativePath

Arguments:

Type:	string	string
Description:	Full path name	Reference directory
Compulsory:	Yes	No
Default:		Current directory

Return type: string

Returns a path relative to the reference directory (argument 2 or current working directory) of the full path name supplied in argument 1.

See also

FullPath

SplitPath

RemoveModelFile

Type	string array
Description	Model path names
Compulsory	Yes
Default	

Return type: string

Uninstalls the model library paths specified in the argument.

RestartTranDialog

Type	real
Description	Initial stop time
Compulsory	Yes
Default	

Return type: real

Opens a dialog box allowing the user to specify a new stop time for a transient analysis. The value is initialised with the argument. The return value is the stop time entered by the user. The user will not be able to enter a value less than that supplied in the argument.

Rms

Arguments:

Type:	real array
Description:	vector
Compulsory:	Yes
Default:	

Return type: real array

Returns a vector of the accumulative rms value of the input. Unlike **RMS1** this function returns a vector which can be plotted.

RMS1

Arguments:

Type:	real array	real	real
Description:	vector	start x value	end x value
Compulsory:	Yes	No	No
Default:		start of input vector	end of input vector

Return type: real

Returns the root mean square value of the supplied vector between the ranges specified by arguments 2 and 3. If the values supplied for argument 2 and/or 3 do not lie on sample points, second order interpolation will be used to estimate y values at those points.

rnd

Arguments:

Type:	real array
Description:	vector
Compulsory:	Yes
Default:	

Return type: real array

Returns a vector with each element a random value between 0 and the absolute value of the argument's corresponding element.

RootSumOfSquares

Arguments:

Type:	real	real	real
Description:	vector	start x value	end x value
Compulsory:	Yes	No	No
Default:		start of input vector	end of input vector

Return type: real array

Similar to **RMS1** function but returns the root of the sum without performing an average.

Scan

Arguments:

Type:	string	string
Description:	string to scan	delimiter
Compulsory:	Yes	No
Default:		

Return type: string array

Splits a character delimited string into components. Returns result as string array. Character used as delimiter may be passed as argument 2. If argument 2 omitted delimiter defaults to a semi-colon. This function is similar to the **Parse** function but with two important differences:

- Empty fields are supported. E.g. in 'BUF04;buf;;Buffers;;' the double semi-colon after 'buf' would return an empty entry in the returned array. The Parse function would ignore it. So:
  ```
  Scan('BUF04;buf;;Buffers;;')
  ```

would return:

```
['BUF04', 'buf', '', 'Buffers', '']
```

- Only a single character is permitted for delimiter. Parse accepts any number.

ScriptName

No Arguments

Return type: string

Returns the name of the currently executing script.

Search

Arguments:

Type:	string array	string array
Description:	list search	items to search in list
Compulsory:	Yes	Yes
Default:		

Return type: real array

Searches a list of strings supplied in argument 1 for the item(s) supplied in argument 2. Function returns a real array of length equal to the length of argument 2. The return value is an array of indexes into argument 1 for the items found in argument 2. If a string in argument 2 is not found, the return value for that element will be -1.

SearchModels

Arguments:

Type:	string
Description:	File system tree to search
Compulsory:	Yes
Default:	

Return type: string

This is a special purpose function designed for use with the model installation system. It returns an array of strings holding pathnames with wildcards of directories containing files with SPICE compatible models. The argument specifies a directory tree to search. The function will recurse through all sub directories of the supplied path.

Note that if the root directory of a large disk is specified, this function can take a considerable time to return. It can however be aborted by pressing the escape key.

SelectColourDialog

Type	string
Description	Initial colour spec.
Compulsory	No
Default	Spec. for BLACK

Return type: string

Opens a dialog box allowing the user to define a colour. The box is initialised with the colour spec. supplied as an argument. The function returns the new colour specification.

SelectColumns

Arguments:

Type:	string array	real	string
Description:	input data	field number	delimiter
Compulsory:	Yes	Yes	No
Default:			';'

Return type: string array

Accepts an array of character delimited strings and returns an array containing only the specified field.

Example

Data input (arg 1):

BUF600X1;Buf;;Buffers;;2,1,4,3

BUF600X2;Buf;;Buffers;;2,1,4,3

BUF601X1;Buf;;Buffers;;2,1,4,3

BUF601X2;Buf;;Buffers;;2,1,4,3

Field number (arg2)

0

Returns:

BUF600X1

BUF600X2

BUF601X1

BUF601X2

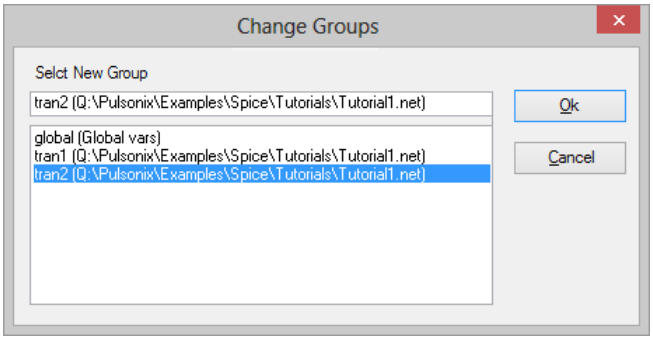
SelectDialog

Arguments:

Type:	string array	string array
Description:	options	list box entries
Compulsory:	Yes	Yes
Default:		

Return type: real array

Opens a dialog box containing a list box. The list box is filled with string items supplied in argument 2. The return value is the index or indexes of the items in the list box selected by the user.



This function is used by a number of the standard menus.

There are a number of options available and these are specified in argument 1. This is an array of strings of length up to 6. The meaning of each element is as follows:

Index	Possible values	Description
0		Dialog box caption
1		Message above list box
2	'Multiple', 'Single'	If 'single', only one item may be selected. Otherwise any number of items can be selected.
3	'Sorted', ''	If 'sorted', items in list are arranged in alphabetical order. Otherwise they are in same order as supplied.
4		Index of item to select at start. Only effective if 'single' selected for index 2. This is an integer but must be entered as a string e.g. '2'.
5		Initial string in edit box
6		Default return value if none selected

The function return value is empty if the user cancels.

Example

```
SelectDialog(['Caption','Message','single','','1'], ['Fred','John','Bill'])
```

Will place strings 'Fred', 'John' and 'Bill' in the list box with 'John' selected initially. The strings will be in the order given (not sorted).

SelectFontDialog

Type	string	string
Description	Initial font spec.	Name of object being edited
Compulsory	No	No
Default	Default font	

Return type: string

Opens a dialog box allowing the user to define a font. The box is initialised with the font spec. supplied as an argument. The function returns the new font specification.

A second argument may be specified to identify the name of the object whose font is being edited. This is so that its font may be updated if the user presses the Apply button in the dialog box.

If the user cancels the box, the function returns an empty vector.

Font specs are strings that provide information about the type face, size, style and other font characteristics. Font specs should only be used with functions and commands that are designed to accept them. The format of the font spec may change in future versions.

SelectRows

Arguments:

Type:	string	string	real	string
Description:	input data	test string	field number	delimiter
Compulsory:	Yes	Yes	No	No
Default:			0	';'

Return type: string array

Accepts an array of character delimited strings and returns an array containing a selection containing the test string at specified field.

Example

Data input (arg 1):

HA-5002/HA;buf;;Buffers;;

HA-5033/HA;buf;;Buffers;;

HA5002;buf;;Buffers;;

HA5033;buf;;Buffers;;

LM6121/NS;buf;;Buffers;;1,2,4,3

MAX4178;buf_5;;Buffers;;

MAX4278;buf_5;;Buffers;;

MAX496;buf_5;;Buffers;;

Test string (arg 2)

'buf'

Field number (arg 3)

1

Returns:

HA-5002/HA;buf;;Buffers;;

HA-5033/HA;buf;;Buffers;;

HA5002;buf;;Buffers;;

HA5033;buf;;Buffers;;

LM6121/NS;buf;;Buffers;;1,2,4,3

SelGraph

No Arguments

Return type: real

Returns 1 if at least one graph is open otherwise 0.

Shell

Type	string	string
Description	Path to executable file	options
Compulsory	Yes	No
Default		

Return type: real array

Runs an external program and returns its exit code

Argument 1

File system path to executable file. This would usually be a binary executable but may be any file that is defined as executable by the operating system.

If an incomplete path is specified, the process executable will be searched in the following locations in the order given:The directory where the Pulsonix binary is located

1. The current directory
2. *windows*\SYSTEM32. *windows* is the location of the Windows directory.
3. windows\SYSTEM
4. The windows directory
5. The directories listed in the PATH environment variable

Argument 2

String array containing one or more of the options defined in the following table:

Option name	Description
'wait'	If specified, the function will not return until the called process has exited.
'command'	Calls OS command line interpreter to execute the command supplied. This can be used to execute system commands such as 'copy' and 'move'.

Return Value

Returns a real array of length 2 as defined below:

Index	Description
0	Process exit code. If the process is still running when this function returns, this value will be 259.
1	Error code as follows
	0 Process launched successfully
	1 Command processor not found. (*command* options specified
	2 Cannot find file
	3 File is not executable
	4 Access denied
	5 Process launch failed
	6 Unknown failure

ShellExecute

Type	string	string	string	string
Description	File	parameters	default directory	verb
Compulsory	Yes	No	No	No
Default		none	current directory	'open'

Return type: string

Performs an operation on a windows registered file. The operation to be performed is determined by how the file is associated by the system. For example, if the file has the extension PDF, the Adobe Acrobat or Adobe Acrobat Reader would be started to open the file. (Assuming Acrobat is installed and correctly associated)

Argument 1

Name of file to process. This can also be the path to a directory, in which case an 'explorer' window will be opened.

Argument 2

Parameters to be passed if the file is an executable process. This should be empty if arg 1 is a document file.

Arguments 3

Default directory for application that processes the file

Argument 4

'Verb' that defines the operation to be performed. This would usually be 'open' but could be 'print' or any other operation that is defined for that type of file.

Return Value

Returns one of the following

Value	Description
'OK'	Function completed successfully
'NotFound'	File not found
'BadFormat'	File format was incorrect
'AccessDenied'	File could not be accessed due to insufficient privilege
'NoAssoc'	File has no association for specified verb
'Share'	File could not be accessed because of a sharing violation
'Other'	Function failed for other reason
'NotImplemented'	Function not implemented on this platform.

sign

Arguments:

Type:	real array
Description:	vector
Compulsory:	Yes
Default:	

Return type: real array

Returns 1 if argument is greater than 0 otherwise returns 0.

SimulationHasErrors

No arguments

Return type: real

Return 1 if the most recent simulation failed with an error. Otherwise returns 0.

Note that the function will return 0 if no simulation has been run or if the simulator has been reset using the Reset command. It will also return 0, if the simulation failed because of a fatal error that caused the simulator process to restart. This occurs when an access violation or floating point exception occurs.

sin

Type	real/complex array
Description	vector
Compulsory	Yes
Default	

Return type: real/complex array

Return sine of argument specified in radians.

sin_deg

Type	real/complex array
Description	vector
Compulsory	Yes
Default	

Return type: real/complex array

Return sine of argument specified in degrees.

Sleep

Type	string
Description	Time in seconds

Compulsory Yes
Default

Return type: real

Executes a timed delay.

Argument 1

Delay in seconds. The function has a resolution of 100mS and so the delay will be integral multiples of that amount.

Return Value

Function returns the value of the argument.

Sort

Type:	string
Description:	string data
Compulsory:	Yes
Default:	

Return type: string array

Performs alphanumeric sort on string array. Result is string array the same length as argument.

SortIdx

Arguments:

Type:	any array	string
Description:	items to sort	sort direction
Compulsory:	Yes	No
Default:		'forward'

Return type: real array

Sorts the items in argument 1 but instead of returning the actual sorted data the function returns the indexes of the sorted values into the original array. The method of sorting depends on the data type as follows:

string	Alphabetic
real	Numeric
complex	Numeric - uses magnitude

If the second argument is 'reverse' the sort is performed in reverse order.

SplitPath

Arguments:

Type:	string
Description:	path
Compulsory:	Yes
Default:	

Return type: string array

Splits file system pathname into its component path. Return value is string array of length 4:

0	Drive including ':'. E.g. 'C:'
1	Directory including prefix and postfix '\'. E.g. "Program Files\Pulsonix\Spice\'
2	Filename without extension. E.g. 'PulsonixSpice'

3 Extension including period. E.g. '.EXE'

sqrt

Arguments:

Type:	real/complex array
Description:	vector
Compulsory:	Yes
Default:	

Return type: real/complex array

Returns the square root of the argument. If the argument is real and negative, an error will result. If however the argument is complex a complex result will be returned.

Str

Arguments:

Type:	any
Description:	
Compulsory:	Yes
Default:	

Return type: string

Returns the argument converted to a string.

StringLength

Arguments:

Type:	string
Description:	input string
Compulsory:	Yes
Default:	

Return type: real

Returns the number of characters in the supplied string.

StrStr

Type	string	string	real
Description	input string	sub string	offset
Compulsory	Yes	Yes	No
Default			0

Return type: real

Locates the sub string in argument 2 in the input string. If found the function will return the character offset of the sub string. If not found the function will return -1.

Argument 1

String to search

Argument 2

Sub-string

Argument 3

Offset into search string where search should begin.

Return Value

Number of characters from start of search string where sub string starts. -1 if substring is not found.

SumNoise

Arguments:

Type:	real	real	real
Description:	vector	start x value	end x value
Compulsory:	Yes	No	No
Default:		start of input vector	end of input vector

Return type: real array

Identical to **RootSumOfSquares** function.

tan

Type	real/complex array
Description	vector
Compulsory	Yes
Default	

Return type: real/complex array

Return tan of argument specified in radians.

tan_deg

Type	real/complex array
Description	vector
Compulsory	Yes
Default	

Return type: real/complex array

Return tan of argument specified in degrees.

Time

Arguments:

Type:	string
Description:	option
Compulsory:	No
Default:	<<empty>>

Return type: string

Returns the current time in the format specified in control panel.

TranslateLogicalPath

Type	string
Description	Symbolic path
Compulsory	Yes
Default	

Return type: string

Converts symbolic path to a physical path.

Argument 1

Symbolic path as described in the “Sundry Topics” chapter of the User's Manual.

Return Value

Returns actual file system path.

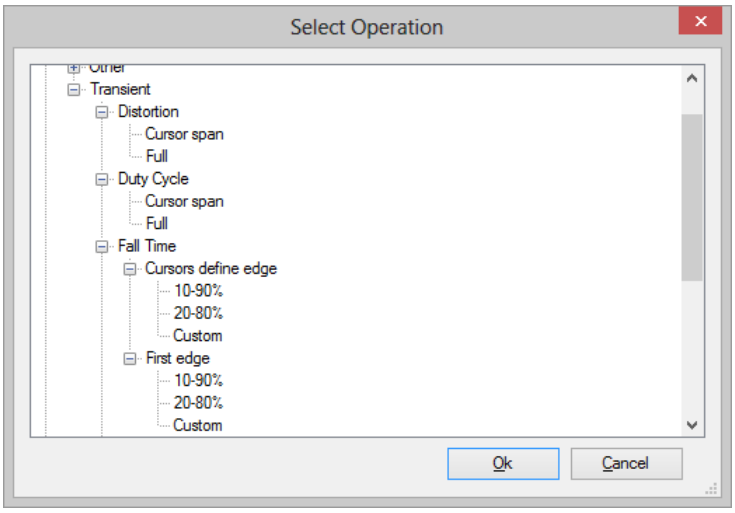
TreeListDialog

Arguments:

Type:	string array	string array
Description:	strings	options
Compulsory:	Yes	No
Default:		['Select Item', '', '0', 'sort', 'false']

Return type: real

Opens the following dialog box allowing the user to specify an item in tree structured list.



Argument 1

Specifies the items to be displayed in the tree list. These are arranged in semi-colon delimited fields with each field specifying a "branch" of the tree. For example, in the above diagram, the item shown as "Full" would be specified as an element of argument 1 as "Measure;Transient;RMS;Full".

Argument 2

An array of strings of max length 5 specifying various other characteristics as defined below:

0	Dialog caption
1	Identifies an item to be initially selected using the same format as the entries in argument 1.
2	Initial expand level. '0' for no expansion, '1' expands first level of tree etc.
3	Items will be alphabetically sorted unless this is set to 'nosort'
4	Items may selected and the box closed by double clicking unless this item is set to 'true'

Return value

Returns index into argument 1 of selected item. If no item is selected, the function returns -1. If the user selects "Cancel" the function returns an empty vector.

TRUE

Arguments:

Type:	string	string
Description:	vector name	option
Compulsory:	Yes	No
Default:		<<empty>>

Return type: real

Returns TRUE (1) if the vector specified by name in argument 1 exists AND is non-zero. If argument 2 is set to 'SearchCurrent', the *current* group as well as the *local* and *global* groups will be searched for the vector, otherwise only the *local* and *global* groups will be searched. See Accessing Simulation Data for an explanation of groups.

Truncate

Arguments:

Type:	real array	real	real
Description:	vector	start x value	end x value
Compulsory:	Yes	No	No
Default:		start of vector	end of vector

Return type: real array

Returns a portion of the input vector with defined start and end points. Interpolation will be used to create the first and last points of the result if the start and end values do not coincide with actual points in the input vector.

Arguments 2 and 3 define the beginning and end of the vector.

Example

Suppose we have a vector called VOUT which was the result of a simulation running from 0 to 1mS. We want to perform some analysis on a portion of it from 250µS to 750µS. The following call to Truncate would do this:

```
Truncate(VOUT, 250u, 750u)
```

If VOUT did not actually have points at 250µS and 750µS then the function would create them by interpolation. Note that the function will not extrapolate points before the start or after the end of the input vector.

Units

Arguments:

Type:	any
Description:	vector or vector name
Compulsory:	Yes
Default:	

Return type: string

Returns the physical units of the argument. Possible return values are

" (meaning dimensionless)

'?' (meaning unknown)

'V'

'A'

'Secs'

'Hertz'

'Ohm'

'Sie'

'F'

'H'

'J'

'W'

'C'

'Vs'

'V^2'

'V^2/Hz'

'V/rtHz'

'A^2'

'A^2/Hz'

'A/rtHz'

'V/s'

See also

PhysType

unitvec

Arguments:

Type:	real
Description:	Number of elements in result
Compulsory:	Yes
Default:	

Return type: real array

Returns a vector consisting of all 1's. Argument specifies length of vector.

UpDownDialog

Arguments:

Type:	string array	string
Description:	strings to sort	box caption
Compulsory:	Yes	No
Default:		'Select Item Order'

Return type: string array

Opens a dialog box to allow the user to rearrange the order of a list of strings.

The box displays the strings given in argument 1 in the order supplied. The user can rearrange these using the up and down arrow buttons. When the user presses OK the function return the strings in the new order. If the user cancels the box the function returns an empty vector.

Val

Arguments:

Type:	any
Description:	input value
Compulsory:	Yes
Default:	

Return type: real/complex

Returns argument converted to a value. The conversion assumes that the string supplied is an expression.

See also

Str

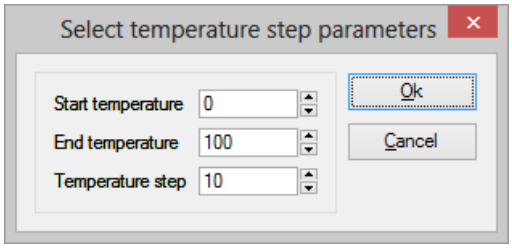
ValueDialog

Arguments:

Type:	real	string	string	string
Description:	Initial edit control values	Edit control labels	Dialog box caption	Special characteristics
Compulsory:	No	No	No	No
Default:	1	empty	empty	none

Return type: real array

Opens a dialog box with up to 10 edit controls allowing numeric values to be entered. The following is an example of dialog box when used for selecting temperature step parameters:



The number of edit controls displayed is determined by the length of the first argument. If this is omitted, all 10 will be displayed. Argument 1 specifies the initial values set in each of the controls.

Argument 2 supplies the text of the label displayed to the left of each edit control. The width of the dialog box will be adjusted to accommodate the length of this text.

Argument 3 specifies the text in the title bar of the dialog box

The value supplied for argument 4 will be treated as the default. All boxes are allowed to vary over a range of -1e18 to +1e18.

The function returns an array representing the user selected value in each box. If cancelled it returns an empty vector.

See also

BoolSelect

EditSelect

RadioSelect

vector

Arguments:

Type:	real
Description:	Number of elements in result
Compulsory:	Yes
Default:	

Return type: real array

Returns a vector with length specified by the argument. The value in each element of the vector equals its index.

See also

UnitVec

VectorsInGroup

Arguments:

Type:	string	string
Description:	group name	option
Compulsory:	No	No
Default:	Current group	

Return type: string array

Returns the names or optionally the physical type of all vectors in the specified group. If argument 2 is present and equal to 'PhysType' the physical type (e.g. 'voltage', 'current', 'time' etc.) of the vectors will be returned otherwise the function will return their names.

WriteConfigSetting

Type	string	string	string
Description	Section	Key	Value
Compulsory	Yes	Yes	No
Default			

Return type: real

Writes a configuration setting. Configuration settings are stored in the configuration file. See ‘Configuration Settings’ in the *User’s Manual* for more information. Settings are defined by a key-value pair and are arranged into sections. The function writes the value in argument three to the specified key and section. If the value is missing, the setting will be deleted.

Argument 1

Section name

Argument 2

Key name

Argument 3

Value to set. Setting will be deleted if this is omitted.

Return Value

Returns TRUE is value was successfully written. Otherwise returns FALSE.

See Also

ReadConfigSettings

WriteIniKey

Type	string	string	string	string
Description	File	Section	Key	Value
Compulsory	Yes	Yes	Yes	No
Default				Empty string

Return type: real

Writes a value to an 'INI' file. See for more information on INI files.

Argument 1

File name. You should always supply a full path for this argument. If you supply just a file name, the system will assume that the file is in the WINDOWS directory. This behaviour may be changed in future versions. For maximum future compatibility, always use a full path.

Argument 2

Section name.

Argument 3

Key name.

Argument 4

Key value

Return Value

Returns 1 if function successful. Otherwise returns 0.

WriteRawData

Type	real/complex array	string	string	string
Description	data	File name	Options	Format of index display
Compulsory	Yes	Yes	No	No
Default				‘ %d’

Return type: string

Writes data to the specified file in a SPICE3 raw file compatible format. See the built in script write_raw_file for an application example. This can be found on the install CD.

The function returns a single string according to the success or otherwise of the operation. Possible values are: ‘success’, ‘nodata’ and ‘fileopenfail’.

WriteRegSetting

Type	string	string	string	string
Description	Key path	Value name	Value to be written	Top level tree
Compulsory	Yes	Yes	Yes	No
Default				‘HKCU’

Return type: string

Writes a string value to the windows registry.

Argument 1

Name of key. This must be a full path from the top level.

Argument 2

Name of value to be read

Argument 3

Value to be written to key

Argument 4

Top level tree. This may be either 'HKEY_CURRENT_USER' or 'HKEY_LOCAL_MACHINE' or their respective abbreviations HKCU and HKLM. Note that you must have administrator rights to write to the HKEY_LOCAL_MACHINE tree.

Return Value

Returns one of three string values as defined below:

Value	Meaning
'Ok'	Function executed successfully
'WriteFailed'	Could not write that value
'InvalidTreeName'	Arg 4 invalid.

XCursor

No Arguments

Return type: real

Returns the horizontal position of the graph measurement cursor. If there is no graph open or cursors are not enabled, the function returns 0.

XDatum

No Arguments

Return type: real

Returns the horizontal position of the graph reference cursor. If there is no graph open or cursors are not enabled, the function returns 0.

XFromY

Arguments:

Type:	real	real	real
Description:	input vector	Y value	Interpolation order (1 or 2)
Compulsory:	Yes	Yes	No
Default:			2

Return type: real array

Returns an array of values specifying the horizontal location(s) where the specified vector (argument 1) crosses the given y value (argument 2). If the vector never crosses the given value, an empty result is returned. The sampled input vector is interpolated to produce the final result. Interpolation order is specified by argument 3.

Note that unlike other functions that use interpolation, **XFromY** can only use an interpolation order of 1 or 2. If a value larger than 2 is specified, 2 will be assumed.

XY

Arguments:

Type:	real array	real array
Description:	y vector	x vector
Compulsory:	Yes	Yes
Default:		

Return type: real array

Creates an *XY Vector* from two separate vectors. An *XY Vector* is a vector that has a reference. The resulting vector will have y values defined by argument 1and the x values (i.e. its reference) of argument 2.

YCursor

No Arguments

Return type: real

Returns the vertical position of the graph measurement cursor. If there is no graph open or cursors are not enabled, the function returns 0.

YDatum

No Arguments

Return type: real

Returns the vertical position of the graph reference cursor. If there is no graph open or cursors are not enabled, the function returns 0.

YFromX

Arguments:

Type:	real	real	real
Description:	input vector	X value	Interpolation order (1 or greater)
Compulsory:	Yes	Yes	No
Default:			2

Return type: real array

Returns an array of values (usually a single value) specifying the vertical value of the specified vector (argument 1) at the given x value (argument 2). If the given x-value is out of range an empty result is returned. The sampled input vector is interpolated to produce the final result. Interpolation order is specified by argument 3.

Chapter 3. Command Reference

Notation

Symbols used:

Square brackets: `[ ]`

These signify a command line parameter or switch which is optional.

Pipe symbol: `|`

This signifies either/or.

Ellipsis: `...`

This signifies 1 or more optional multiple entries.

Fonts

Anything that would be typed in is displayed in `a fixed width` font.

Command line parameters are in *`italics`*.

Case

Although upper and lower cases are used for the command names, they are NOT in fact case sensitive.

Examples

```
OpenGroup [ /text ] [ filename ]
```

Both `/text` (a switch) and *`filename`* (a parameter) are optional in the above example.

So the following are all legitimate:

```
OpenGroup
OpenGroup /text
OpenGroup run23.dat
OpenGroup /text output.txt
```

```
DelCrv curve_number...
```

1 or more *curve_number* parameters may be given.

So the following are all legitimate:

```
DelCrv 1 2 3
DelCrv 1
```

Command Reference

Abort

Aborts the current simulation. Abort performs the same action as Pause followed by Reset. It stops the current run and then deletes all data associated with it except for any simulation vectors.

Note that this command can only be executed by an assigned key or menu with the direct execution option specified.

About

About

Displays the *about box* which provides version and copyright information.

AddCurveMarker

AddCurveMarker *curve-id division x-position y-position label* [*length* [*angle*]]

Adds a curve marker to the currently selected graph sheet. A curve marker is a graph annotation object and its purpose is to label a curve for the purposes of identification or to highlight a feature. See “Graph Objects” for more information.

curve-id Id for curve to which marker will be attached.

division Division of curve if curve-id refers to a curve group created by a multi-step run. Divisions are numbered from 0 up to 1 minus the number of curves in the group. For single curves set this to zero.

x-position X-axis location of marker.

y-position Y-axis location of marker. This is only used if the curve is non monotonic and has more than one point at x-position. The marker will be placed at the point on the curve with the y-axis value that is nearest to y-position.

label Label for marker. This may use symbolic values enclosed by '%'.

length Length of marker line in *view units*. If omitted *length* defaults to 0.1.

angle Angle of the marker line in the *view co-ordinate*. Default is 45°

AddFreeText

AddFreeText [/font *font-name*] [/align *align*] *text* [*x-pos* [*y-pos*]]

Adds a free text item to the currently selected graph sheet. Free Text is a graph annotation object.

font-name Name of font object to be used for text object. This must either be a standard font (as listed in menu File|Options|Font...) or a font created with the CreateFont command.

align Integer that specifies alignment of text. Possible values:

0	Left bottom
1	Centre bottom
2	Right bottom
4	Left base line
5	Centre base line
6	Right base line
8	Left top
9	Centre top
10	Right top
12	Left middle
13	Centre middle
14	Right middle

text The text to be displayed

x-pos x-co-ordinate of the text in *view units*. Default = 0.5

y-pos *y*-co-ordinate of the text in *view units*. Default = 0.5

AddGraphDimension

```
AddGraphDimension [/vert] [/label label] curve-id1 [pos1 [curve-id2 [pos2]]]
```

Adds a dimension object to a graph. The dimension object is not yet supported by the GUI.

/vert	If present, a vertical dimension is displayed, otherwise it will be horizontal.
label	Text to add to the dimension object
curve-id1	Id of first curve
pos1	Initial position on curve of dimension. X value if horizontal, otherwise a Y value
curve-id2	Id of second curve
pos2	Initial position on second curve of dimension. X value if horizontal, otherwise a Y value

AddLegend

AddLegend [/autoWidth] [/font *fontName*] [*label* [*x-pos* [*y-pos* [*width* [*height*]]]]]

Adds a legend box to the currently selected graph. A “Legend Box” is a graph annotation object which consist of a rectangle containing a list of curve labels

/autoWidth	If specified, the width of the box will be adjusted automatically according to its contents.
fontName	Specifies a font to use for the text contained in the box. Must be either a standard font name or one created using the CreateFont command.
label	This is the text that will copied to each entry. To be meaningful this must contain a symbolic value enclosed by '%'. The default value for *label* if omitted if %DefaultLabel%. This will result in the curves name and measurements being displayed in the legend box. Some alternatives are: %Curve:Label% displays just the label with no measurements %Curve:Measurements% displays just the measurements %Curve% displays the curve's ID only %Curve:Label%/%Curve:YUnit% displays the curve name and y-axis units
x-pos	X position of box in view units. If the value is 1.0 or greater, the box will be placed such that its left hand edge is to the right of the graph's grid area. Default = 0
y-pos	Y position of box in view units. If the value is 1.0 or greater, the box will be placed such that its bottom edge is above the graph's grid area. Default = 1
width	Physical width of box in mm. (For CRT monitors this won't be exact. They are typically assumed to be 75 pixels/inch so 1mm is approx. 3 pixels). Note that this value will be ignored if /autowidth is specified. Default = 50.
height	Physical height of box in mm. (See notes above wrt CRT monitors)

AddLegendProp

AddLegendProp *curve_id property_name property_value*

Adds a property to a graph legend. Legend properties are generally used to display measurement information for a curve. Their name and value is displayed below a curve's legend (or label).

curve_id	Curve Id. Curve id is returned by the functions **GetSelectedCurves, GetAxisCurves** and **GetAllCurves**
property_name	Name of property. May be any string and may contain spaces.
property_value	Value of property. May be any string and may contain spaces.

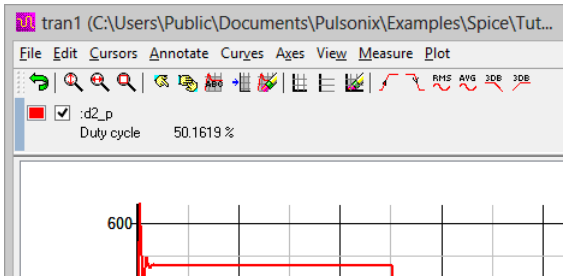
Example

The following is extracted from the script **curve_duty** which displays duty cycle for selected curves.

```
let curves=GetSelectedCurves()
let numCurves = length(curves)
...
for idx=0 to numCurves-1
```

Script lines to retrieve duty cycle ...

```
    AddLegendProp {curves[idx]} "Duty cycle" {duty_cycle}
next idx
```



A typical result is displayed above. In this example the property name is "Duty Cycle" and its value is "50.1619%"

AddTextBox

AddTextBox [/font *font-name*] *text* [*x-position* [*y-position*]]

Adds a Text Box to the currently selected graph. A text box is an item of text enclosed by a border.

font-name	Name of font to be used for text. This must either be a built in font or one created using CreateFont.
text	Text to be displayed in the box. This may use symbolic value enclosed by '%'. The following are meaningful for Text Box objects:
%Date%	The date when the object was created
%Time%	The time when the object was created
%Version%	The name and current version of the program
The *x-position*	The x position of the box in *view units*
y-position	The y position of the box in *view units*

Arguments

```
Arguments [ argument_list ]
```

Declares arguments for a script. Full details for passing arguments to scripts are given in "Script Arguments".

argument_list	List of arguments to be used in the script in the order in which they are passed. Arguments that are *passed by reference* should be prefixed with '@'.

Cd

```
Cd [directory_name]
```

Cd is almost identical to the DOS cd or chdir commands. It changes the current directory to that specified. Unlike the DOS command, however, it will also change the current drive if it is included in the directory name. If no directory name is specified, the current directory will be displayed.

ChooseColour

```
ChooseColour
```

Opens colour selection dialog box.

ClearMessageWindow

```
ClearMessageWindow
```

Clears the command shell message window

Close

```
Close graph
```

Closes the selected graph window.

CloseGraphSheet

```
CloseGraphSheet
```

Closes the current tabbed sheet in the selected graph window. If the window has only one sheet, the whole window will be closed.

ClosePrinter

```
ClosePrinter [ /abort ]
```

ClosePrinter is one of a number of commands and functions used for non-interactive printing. This is explained in "Non-interative and Customised Printing". Printing sessions are started with **OpenPrinter** after which print output commands such as **PrintGraph** may be called. The session is terminated with **ClosePrinter** which actually initiates the printing activity. If the /abort switch is specified, the print job is terminated and no print output will be produced.

See also

NewPrinterPage

OpenPrinter

PrintGraph

GenPrintDialog

GetPrinterInfo

CollectGarbage

CollectGarbage

Deletes temporary vectors. This command is only needed for scripts running endless or very long loops. The simulator creates temporary vectors when calculating vector expressions. These do not get deleted until control is returned to the command line. In the case of a script that calculates many expressions, it is possible for the memory used by the temporary vectors to become excessive. Calling CollectGarbage at regular intervals will resolve this problem.

CreateFont

CreateFont font-name font-base

Creates a new font object based on an existing font. The name given to the font can be used to specify the font for some graph annotation objects. Once CreateFont is called, its name will be displayed in the list displayed when the File|Options|Font... menu is selected.

font-name Name of new font

font-base Name of font to be used to set initial properties. May be any font listed in the menu File|Options|Font... or one of the following: Standard, StandardMedium or StandardLarge.

CreateGroup

CreateGroup [/title title] label

Creates a data group. All vectors (or variables) are organised into groups. Each simulation run creates a new group and all data for that simulation is placed there.

label Base name of group. The actual group name will be appended by a number to make it unique. The new group will become the *current* group. To find the name actually used, you can call the function Groups immediately after calling this command. The first element of Groups (i.e. (Groups())[0]) is always the current group.

title Optional title. This will be displayed in the box displayed when selecting a Change Data Group... menu. It is also returned by a call to Groups('title')

CreateToolBar

CreateToolBar *window_name toolbar_name* [*caption* [*visibility*]]

Creates a new empty toolbar. To add buttons to the toolbar use command "DefineToolBar"

window_name Name of window where toolbar is to reside. Must be one of:

CommandShell	Command shell window
Graph	Graph windows

toolbar_name User assigned name for toolbar. You can use any name that doesn't clash with a pre-defined toolbar name as defined in the table below. The name must not contain spaces.

Pre-defined toolbars:

CommandShellMain	Command Shell toolbar
GraphMain	Graph window toolbar

This name is used to reference the tool bar in the DefineToolBar and SetToolBarVisibility commands.

Caption	Optional caption for toolbar. This is displayed in the caption bar of the toolbar that is visible when the toolbar is 'undocked'.
Visibility	Specifies when the tool bar is visible. This can be subsequently changed using the SetToolBarVisibility command. Possible values are:
always	toolbar is always visible
never	toolbar is never visible

CreateToolButton

CreateToolButton [/toggle] [/class *class_name*] *name graphic* [*hint*]

Creates or redefines a tool bar button. This command creates the properties of the button but not the command it executes when it is pressed. To define the command, use DefButton.

/toggle	If specified, the button will have a toggle action and will have two commands associated with it. One command will be executed when the button is pressed and another when it is released. The 'Wire' pre-defined button is defined in this manner
class_name	This is used with the function GetToolButtons to select buttons according to their function. Set this value to 'component' if you wish the button to be displayed in the GUI which selects component button.
name	Name of button. This may be one of the pre-defined types described in "DefineToolBars" in which case this command will redefine its properties. You may also specify a new name to create a completely new button.
graphic	Graphical image to be displayed on the button. This may be one of the pre-defined images listed in DefineToolBar or you may use a user defined image specified in a file. The file must be located at the following location: Windows: …\support\images where …\ is the top level directory in the tree. The file may use windows bitmap (.bmp), portable network graphic (.png) or JPEG (.jpg) formats. The PNG format supports masks and this format must be used if transparent areas are needed in the graphic. If no mask is found in the graphic file, one will be created which will make the area outside the outer perimeter transparent.
hint	Text that describes the operation of the button. This will be displayed when the user passes the mouse cursor over the button.

CursorMode

CursorMode on | off |toggle | step | stepref | stepShift | stepRefShift

Switches cursor mode of selected graph. In *cursor mode*, two cursors are displayed allowing measurements to be made. See the User's manual for more information on cursors.

on	Switch cursors on

off	Switch cursors off
toggle	Toggles on|off
step	Step cursor to next curve
stepref	Step reference cursor to next curve
stepShift	Steps cursor to next curve within a group. Curves are grouped - for example - for Monte Carlo runs.
stepRefShift	Steps reference cursor to next curve within a group. Curves are grouped - for example - for Monte Carlo runs.

Curve

Curve [/autoXlog]

[/autoYlog]
[/autoAxis
[/axisId *axis_id*]

[/bus type]
[/coll]
[/dig]
[/icb clipboard_index]
[/loglog]
[/name curve_name]
[/newAxis]
[/newGrid]
[/newSheet]
[/newWindow]
[/select]
[/title title]
[/xauto]
[/xdelta x_grid_spacing]
[/xl x_low_limit x_high_limit]

[/xlabel x_label_name]
[/xlog]
[/xunit x_unit_name]
[/yauto]
[/ydelta y_grid_spacing]
[/yl y_low_limit y_high_limit]
[/ylabel y_label_name]
[/ylog]
[/yunit y_unit_name]

[y_expression]
[x_expression]

Curve can be used to add a new curve to an existing graph created with Plot or to change the way it is displayed.

/autoXlog	Only effective when graph sheet is empty. If specified, the x- axis will be logarithmic if the x-values are logarithmically spaced.
/autoYlog	Only effective when graph sheet is empty. Same as /autoxlog except that if x-values are logarithmically spaced, the Y axis will be logarithmic
/autoAxis	If specified, the new curve will be added to a compatible axis according to its physical units i.e Voltage, Current etc. The rules used are as follows: If the currently selected axis or grid (shown by black axis line) has the same units as curve to be plotted or if it has undefined units (designated by a '?' on label), that axis will be used.

	If any other axis or grid has compatible units (i.e same as curve or undefined) that axis will be used.
/axisId axis_id	If specified, the new curve will be added to a y-axis with the id specified by axis_id. Axis id is returned by the functions GetAllYAxes, GetCurveAxis and GetSelectedYAxis. These are documented in the “Script Reference Manual”. This is available as a PDF file on the install CD. A hardcopy version is also available for an additional charge.
/bus type	If specified, the new curve will be plotted on a digital axis and will be plotted as a bus curve. type may be ‘hex’, ‘dec’ or ‘bin’ specifying hexadecimal, decimal or binary display respectively.
/coll	Does nothing. For compatibility with version 3.1 and earlier.
/*dig*	If specified, new curve will be plotted on new digital axis. Digital axes are stacked on top of main axes and are sized and labelled appropriately for digital waveforms.
/icb clipboard_index	Specifies the internal clipboard as the source of the curve data. clipboard_index is a value of 0 or more that indicates which curve in the internal clipboard is to be used. The function may be used to determine the number of curves available. The maximum acceptable value for clipboard_index is thus one less than the value returned by HaveInternalClipboardData.
/loglog	Only effective when graph sheet is empty. Forces both y and x axes to be
/name curve_name	If specified, curve will be named curve_name.
/*newAxis*	If specified, the new curve will be plotted on a new y-axis.
/*newGrid*	If specified, the new curve will be plotted on a new grid.
/*newSheet*	Does nothing. Included for compatibility with Plot command.
/*newWindow*	Does nothing. Included for compatibility with Plot command.
/*select*	If specified, the new curve will be selected.
/title title	Does nothing. Included for compatibility with Plot command.
/*xauto*	Flag. Use automatic limits for x-axis. If this appears after a /xl specification /xauto will override it and vice-versa.
/*xdelta*	Specify spacing between major grid lines on x-axis. Followed by...
x_grid_spacing	Real. For default spacing use '0'.
/xl	Use fixed limit for x-axis. Followed by ...
x_low_limit	Real. Lower limit of x-axis.
x_high_limit	Real. Higher limit of x-axis.
/*xlabel*	Specify label for x-axis. Followed by...
x_label_name	Text String. Label name. If it contains spaces, whole string must be enclosed in quotes ("").
/*xlog*	Only effective when graph sheet is empty. Forces logarithmic x-axis.
/*xunit*	Specify units for x-axis (Volts, Watts etc.). Followed by ...
x_unit_name	Text string. Unit name. If it contains spaces, the whole string must be enclosed in quotes (""). You should not include an engineering prefix (m, K etc.).
/*yauto*	Flag. Use automatic limits for y-axis. If this appears after a /yl specification /yauto will override it and vice-versa.

/ydelta	Specify spacing between major grid lines on y-axis. Followed by...
y_grid_spacing	Real. For default spacing use '0'.
/yl	Use fixed limit for y-axis. Followed by ...
y_low_limit	Real. Lower limit of y-axis.
y_high_limit	Real. Higher limit of y-axis.
/ylabel	Specify label for y-axis. Followed by...
y_label_name	Text String. Label name. If it contains spaces, whole string must be enclosed in quotes ("").
/yunit	Specify units for y-axis. Followed by ...
y_unit_name	Text string. Unit name. Other comments as for x unit name.
y_expression	Text string. Expression describing curve to be added to graph.
x_expression	Text string. Expression describing x values of curve defined by y expression. If omitted, reference of y_expression will be used.

CurveEditCopy

CurveEditCopy *curve-id1* [*curve-id2* ...]

Copy specified curves to the internal clipboard. Curves so copied may be subsequently plotted using the command xxxxxx with the /icb switch.

curve-idn	Id of curve. A number of functions return this value including Default_XREF_styleparatext(page 79).

See Also

Default_XREF_styleparatext

Default_XREF_styleparatext

DefButton

DefButton [/immediate] *button_name command* [*up_command*]

Defines the command executed when a button is pressed.

/immediate	If specified, the command will be enabled for immediate execution. That is the command will be executed immediately even if another command - such as a simulation run - is currently in progress. This will only be accepted when the command specified is one of a small number of built-in command enabled for immediate execution. For the list of commands, see “DefMenu”. You may not call a script if immediate execution is specified.
button_name	Name of button. Either a pre-defined button as listed in “DefineToolBar” or a new button created with CreateToolButton”
command	Command to be executed when the button is pressed.

DefItem

Defitem /bypos *menuname itemname* [*command_string* [*option_flag* [*when_to_enable*]]]

Defitem is used to define custom menu items. It can also be used to delete menu items.

DefItem is included for backward compatibility. New applications should use **DefMenu** which can also define multi level menus and popup menus. To delete menus, use **DelMenu**.

/bypos	Only effective for deleting menu items. (If command string is absent, the menu item is deleted - see below). If this switch is

specified the item is identified by its order in the menu list with the first item being '0' .In this case *itemname* will be assumed to be a number. Note that deleting by position is the only method of deleting menu separators.

menuname The name of the menu as it appears on the menu bar. If the menu name already exists the new menu item will be added to it. If not a new menu will be created. Underlined characters - as used to signify a keyboard selection letter - should be preceded with a '&'.

itemname The name of the item in the menu. If it already exists its function will be redefined. If not a new one will be created at the bottom of the list. If the /bypos switch is specified, *itemname* should be a number.

command_string A command line command or commands to be executed when the menu item is selected. Multiple commands must be separated by semi-colons (';'). Unless the command string has no spaces, it must wholly enclosed in double quotation marks ("). If omitted the menu item will be deleted.

option_flag A number between 0 and 5 to specify the manner in which the command is executed. These are as follows:

0. Default. Command is echoed and executed. Any text already in command line is overwritten.

1. Command is placed in command line but is not executed. Any text already in command line is overwritten.

2. Command is appended to existing text in command line then executed.

3 Command is appended to existing text in command line but is not executed.

4. Same as 0.

5. Immediate mode. Command is executed immediately even if another operation - such as a simulation run - is currently in progress. For other options the command is not executed until the current operation is completed. Only a few commands can be assigned with this option. These are:
DefItem
DefKey
Echo
Let
Pause
Quit
ScriptAbort
ScriptPause
ScriptResume
Shell

6. Item is a separator i.e. a line separating menu items in order to group them together.

when_to_enable A boolean expression specifying under what circumstances the menu should be enabled. (The menu text turns grey when disabled). If omitted the menu will always be enabled. See DefMenu for further details.

If only the menuname and itemname are specified, the menu item of that description will be deleted.

Examples

```
DefItem &File "Edit File" "shell { 'notepad'&' '&getfile('All
Files\*') }"
```

defines a new menu item to call the notepad editor.

```
DefItem &File "Edit File"
```

deletes the menu item specified

```
DefItem /bypos &File 6
```

Deletes the seventh item in the &File menu

```
DefItem &File " " " " 6
```

Adds a separator (a horizontal line) to the bottom of the &File menu

See Also:

User Defined Key and Menu Definitions.

DefineToolBar

DefineToolBar *toolbar_name button_defs*

Defines the buttons for a user defined toolbar created using CreateToolBar. To define the buttons for a pre-defined toolbar, the associated option setting must be set using the command Set.

toolbar_name — Name of toolbar. This must be a toolbar created using CreateToolBar .

button_defs — Semi-colon delimited list of button names to add to the toolbar. Buttons may either be one defined using CreateToolButton or one of the pre-defined types shown in the table below. The '-' character may also be used to specify a spacer.
Pre-defined buttons:

Button Name	Graphic	Function
AddCurve	newcurve.bmp	Add Curve
AddFourier	newfourier.bmp	Fourier...
CalcAveragePower	avg.bmp	Display Average Power/Cycle
CalcFall	falltime.bmp	Display Fall Time
CalcHighPass3db	3dbhighpass.bmp	Display -3dB Point (High Pass)
CalcLowPass3db	3dblowpass.bmp	Display -3dB Point (Low Pass)
CalcRise	risetime.bmp	Display Rise Time
CalcRMS	rms.bmp	Display RMS/Cycle
Capacitor	cap.bmp	Place Capacitor
Copy	copy.bmp	Duplicate
Delete	erase.bmp	Cut
DeleteAxis	delgrid.bmp	Delete Axis/Grid
DeleteCurve	delete.bmp	Delete Curve
GraphClose	fileclose.bmp	Close Graph
GraphOpen	fileopen.bmp	Open Graph
GraphSave	filesave.bmp	Save Graph
HideCurves	hide.bmp	Hide Selected Curves
MoveCurve	movecurve.bmp	Move Curve to Selected Axis/Grid
NewAxis	newaxis.bmp	New Axis
NewGrid	newgrid.bmp	New Grid
Options	options.bmp	Options
Print	print.bmp	Print
ShowCurves	show.bmp	Show Selected Curves
SimPause	pause.bmp	Pause Simulation
SimRunNetlist	run.bmp	Run Netlist
SimRunSchem	run.bmp	Run Schematic
TitleCurve	curvetitle.bmp	Change Curve Name
UndoZoom	undo.bmp	Undo Zoom
ZoomFull	zoomfull.bmp	Fit Window
ZoomIn	zoomin.bmp	Zoom In
ZoomOut	zoomout.bmp	Zoom Out
ZoomRect	zoomrect.bmp	Zoom Box
ZoomXAuto	zoomwidth.bmp	Fit Width
ZoomYAuto	zoomheight.bmp	Fit Height

The graphic images for all pre-defined buttons are built-in to the program, but the image files from which they were created can be replaced in the \support\images folder..

See Also

DefButton, SetToolBarVisibility, GetToolButtons

DefKey

DefKey *Key_Label Command_string* [*option_flag*]

DefKey is used to define custom key strokes.

Key_Label Code to signify key to define. See table below for list of possible labels. All labels may be suffixed with one of the following:

:GRAPH	Key defined only when a graph window is currently active
:SHELL	Key defined only when the command shell is currently active.

If no suffix is provided the key definition will be active in all windows.

Command_string A command line command or commands to be executed when the specified key is pressed. Multiple commands must be separated by semi-colons (';'). Unless the command string has no spaces, it must wholly enclosed in double quotation marks (").

option_flag A number between 0 and 5 to specify the manner in which the command is executed. These are as follows:

0. Default. Command is echoed and executed. Any text already in command line is overwritten.

1. Command is placed in command line but is not executed. Any text already in command line is overwritten.

2. Command is appended to existing text in command line then executed.

3. Command is appended to existing text in command line but is not executed.

4. Same as 0.

5. Immediate mode. Command is executed immediately even if another operation - such as a simulation run - is currently in progress. For other options the command is not executed until the current operation is completed. Only a few commands can be assigned with this option. These are

DefItem
DefKey
Echo
Let
Pause
Quit
ScriptAbort
ScriptPause
ScriptResume
Shell

Note, the **Let** command can be used to set a global variable which can then be tested in running script. This is a convenient method of providing user control of script execution.

Valid key labels:

Function keys :

F1

F2

F3

F4

F5

F6

F7

F8

F9

F10

F11

F12

INS Insert key

DEL Delete key

HOME Home key

END End key

PGUP Page up key

PGDN Page down key

LEFT ←

RIGHT →

UP ↑

DOWN ↓

TAB Tab key

BACK Back space

ESC Escape key

NUM1 Keypad 1

NUM2 Keypad 2

NUM3 Keypad 3

NUM4 Keypad 4

NUM5 Keypad 5

NUM6 Keypad 6

NUM7 Keypad 7

NUM8 Keypad 8

NUM9 Keypad 9

NUM0 Keypad 0

NUM* Keypad *

NUM/ Keypad /

NUM+ Keypad +

NUM- Keypad -

NUM. Keypad .

_SPACE Space bar (must always be shifted - see below)

All letter and number keys i.e.

A to Z and 0 to 9 referred to by letter/number alone.

Shifted keys

Any of the above prefixed with any combination of 'S' for shift, 'C' for control or 'A' for alt. Note that in windows, the right hand ALT key performs the same action as CONTROL-ALT.

Notes

Unshifted letter and number key definitions will not function when a text edit window is active. Space bar definitions must always be shifted.

The same codes can be used for menu short cuts. See **DefMenu** command.

Examples

The mc_histo script uses the following command:

```
DefKey CQ "Let /ne global:abortHisto = 1" 5
```

This defines control Q to set a global variable to 1 to force abort on the histogram process. It resets the key as follows:

```
DefKey CQ ""
```

The definition for F12 to zoom in a graph is

```
DefKey F12:GRAPH "SizeGraph 0 0 0.8 0.8" 4
```

This definition only functions when a graph window is active.

Note that the key definition will be lost when Pulsonix Spice is quitted. To make a key or menu definition permanent you can place the command to define it in the startup file. To do this, select command shell menu **File|Scripts|Edit Startup** and add the line above.

DefMenu

DefMenu [/immediate] [/shortcut *key_code*] [/insert] [/append] *menuname command_string when_to_enable*

Defines custom menu. Supersedes DefItem

/immediate — Immediate mode. Command is executed immediately even if another operation - such as a simulation run - is currently in progress. For other options the command is not executed until the current operation is completed. Only a few commands can be assigned with this option. These are

DefItem
DefKey
Echo
Let
Pause
Quit
ScriptAbort
ScriptPause
ScriptResume
Shell

/shortcut *key_code* — Specify key or key combination to activate menu. Key description is placed on right hand side of menu item. For list of possible values see **DefKey** command. Note that **DefKey** has precedence in the event of the key or key combination being defined by both **DefKey** and **DefMenu**.

/insert — Command isn't executed but the text of the command is placed in the command line overwriting any existing text.

/append — Command isn't executed but the text of the command is placed in the command line appended to existing text

menuname — Composed of strings separated by pipe symbol : '|'. First name must be one of the following:

SHELL	Command shell menu
GRAPH	Graph popup menu

LEGEND	Popup menu in graph "legend panel" (Between toolbar and main graph drawing area)

For SHELL menu, this must be followed by two or more names separated by '|' . The first is the menu name as it appears on the menu bar. The second can be the name of a menu item (which is actioned when selected) or a sub menu containing menu items or further sub menus. Sub menus can be nested to any level.

GRAPH and LEGEND must be followed by at least one name. Sub menus may also be defined for these.

To define a menu separator use the item text "-"

Note that if any of the menu name contains spaces it must be enclosed in quotation marks.

See examples below.

when_to_enable	A boolean expression specifying under what circumstances the menu should be enabled. (The menu text turns grey when disabled). If omitted the menu will always be enabled. The expression may contain the following values:
GraphOpen	TRUE when there is at least one graph window open.
SimPaused	TRUE when the simulator has been paused.
SimRunning	TRUE when the simulator is running.
CircuitLoaded	TRUE when a circuit has been loaded to the simulator. (This happens when ever a simulation is run. A circuit can be unloaded with the Reset command).
LiveMode	TRUE when a command has not completed.
Never	Always FALSE i.e menu permanently disabled.

These values can be combined with the operators:

&&	logical AND
||	logical OR
==	equals
!=	not equal
!	NOT

Parentheses may also be used.

Note that this expression is not related to vector expressions or the expressions that can be used in netlists or the command line.

Examples

The following are definitions for some of the standard menus. You can see them all by executing the **ListStdMenu** command

Print graph command shell menu

DefMenu "Shell|&File|&Print ..." "gen_print /ne" "GraphOpen && !LiveMode"

New Axis on the graph popup menu.

DefMenu "Graph|New A&xis" "NewAxis /ne" "!liveMode"

Separator in graph popup

DefMenu "Graph|-"

Del

Del *filename*

Deletes the specified file. Wildcards may be used for *filename* e.g. *.*. '*' matches any sequence of zero or more characters. '?' matches a single character. Any file matching the specification will be deleted.

DelCrv

DelCrv *curve_id ... |curve name ...*

Deletes the specified curve or curves on the selected graph. *curve_id* is returned by the functions **GetSelectedCurves**, **GetAxisCurves** and **GetAllCurves.**

Optionally a curve name may be specified. This must be the whole text of the curve legend. It is the value returned by the **GetCurves** function.

DeleteAxis

DeleteAxis *axis_id*

Deletes the specified axis.

axis_id	Axis id as returned by functions **GetAllYAxes** , **GetSelectedYAxis** or **GetCurveAxis**.

Note that an axis may only be deleted if it is empty i.e. has no attached curves. Also the main axis may not be deleted.

DeleteGraphAnno

DeleteGraphAnno *object-id*

Deletes a graph annotation object such as a curve marker or legend box. See "Graph Objects" for details on graph annotation objects.

object-id	Id of object to be deleted.

DelGroup

DelGroup [/all] [/cleanup] [/nodelete] *groupname2 ...*

/*all*	If specified all groups except the user group are destroyed.
/*noDelete*	Inhibits delete of associated temporary data file. This file will only be deleted any way if the option variable DataGroupDelete is set to OnDelete.
/*cleanup*	Specify this switch if the associated data file is going to be reused as it may speed up the read operation especially if the data was created by a simulation that was paused. If the file will be deleted then this switch has no benefit but will do no harm other than to slow the execution of this command a little.

Deletes specified groups.

See Also

CreateGroups

Function "Groups"

DelLegendProp

DelLegendProp *curve_id property_name*

curve_id — Id of curve which possesses property. Curve id is returned by the functions **GetSelectedCurves**, **GetAxisCurves** and **GetAllCurves**

property_name — Name of property to be deleted. The function **GetLegendProperties** returns legend properties owned by a specified curve.

DelMenu

DelMenu [/bypos *position*] *menuname*

Deletes specified menu

position — The menu to be deleted is identified by its position. The first item in the menu is at position zero.

menuname — Composed of strings separated by pipe symbol : '|'. First name must be one of the following:

SHELL	Command shell menu
GRAPH	Graph popup menu
LEGEND	Popup menu in graph "legend panel"

The remaining strings identify the menu and item names. See **DefMenu** for details on menu names.

Discard

Discard [*groupname*]

Frees up memory used for vectors. This does not destroy the vectors, just removes any copies that reside in RAM. The data is always stored on disc and can be recovered to RAM when needed.

groupname — Name of group whose data is to be discarded. Use current group if omitted.

Notes

It is rare that this command is needed but mat be useful if you are running long simulations and the data generated is so large that a great deal of disk swapping is taking place.

The vectors created by the simulator are initially stored in a file. If they are needed - usually for plotting a graph - the data is copied to memory. Once the data has been copied to memory, it will stay there until the group to which the vector belongs is destroyed. Simply closing the graph that used the data will not free up the memory as it is assumed that the data may be needed again and the process of reading from the disk can be time consuming. If the data is very large it will consume a lot of memory which can have adverse consequences.

The discard command deletes the data stored in memory for all vectors in the specified group. It does not delete the vectors altogether as they are still stored on disc in the temporary file. After discarding a group, it is still possible to plot all vectors that it contains.

Display

Display [*groupname1* [*groupname2* ...]]

Displays list of all vectors in specified groups or current group by default.

See Also

Expressions

Echo

Echo [/file|/append *filename*] /page /box *text*

Echoes *text* to the message window or to a file

/file filename	If present text is output to filename. If filename exists, it is overwritten
/*append filename*	If present text is appended to filename. If filename does not exist, it is created.
/*page*	Prefixes output with a ASCII form feed character
/*box*	Text is output inside a box composed of asterix characters. This is useful for titles and headings. Currently only works correctly when used with /file or /append.

EditColour

EditColour *colour-name colour-spec*

Changes the spec for the named colour object.

colour-name	Name of colour object. This can be any of the names returned by the GetColours() function. (These are listed when the menu File|Options|Colour... is selected.)
colour-spec	Text string that defines the colour. The functions GetColourSpec() and SelectColourDialog() return colour spec values.

EditFile

EditFile *filename*

Opens an external text editor to edit specified file. The path of the text editor may be specified by the "Editor" option which may be set in the File Locations tab of the options dialog box. (**File|Options|General...**). This is "notepad" by default.

EditFont

EditFont *font-name font-spec*

Changes the spec for the named font object.

font-name	Name of font object. This can be any of the names returned by the GetFonts() function. (These are listed when the menu File|Options|Font... is selected.)
font-spec	Text string that defines the font. The functions GetFontSpec and SelectFontDialog return font spec values.

ExecuteMenu

ExecuteMenu *menu-item*

Executes the specified menu.

menu-item	Menu definition as described in DefMenu command.

Execute

Execute *command*

Run the script or command *command.*

Scripts are usually run by simply entering their name in the same way as a command is entered. However, the script is executed slightly differently if run using the **Execute**

command. If a script is called from another script in the normal way, the called script is read in and parsed before the main script is executed. If the **Execute** command is used, the called script is not read in until and unless the **Execute** command is actually executed. This has two main applications.

1. The name of the called script is not known initially. This is the case with the menu item **File|Script|Run Script...** . The script name is selected from a dialog box. The **Execute** command is used to implement this menu item.
2. The called script is very long and is not always called by the calling script. It may take some time to read in and parse the called script. This time would be wasted if the script is not actually called.

Avoid using **Execute** if a script is called within a loop. The script would be read in and parsed each time around the loop which is very inefficient.

Focus

Focus [/named *window_name*] [graph]

/*named window_name*	If specified the window of the given name will be given input focus. The name of the window is the text in the title bar with "(Selected)" stripped off. Window name is also returned by **GetWindowNames** function.
graph	Currently selected graph window receives input focus.

See Also

GetWindowNames

FocusShell

FocusShell

Selects the Command Shell and assigns it keyboard focus.

Font

Font

Opens the font selection dialog box.

GraphZoomMode

GraphZoomMode X|Y

Specifies mode of next mouse zoom operation

X	Only X axis will be zoomed
Y	Only Y axis will be zoomed

All subsequent zoom operations will be applied to both axes.

Help

Help [/file *filename*] [/contents] [/context *context_id*] [*topic*]

Opens the **Pulsonix Spice** help system.

filename	If specified, help will be obtained from *filename*. Otherwise help file will be SIMULATOR.HLP
/contents	Display contents page. Overrides /context and *topic*.

context_id Opens specific topic indentified by an integer. This is used by some internal scripts but is not supported for user application.

topic If specified, help system will display page relating to *topic*. If *topic* does not exist, a list of available topics will be displayed.

Example:

To display help on the .tran simulator directive type:

```
Help .tran
```

HideCurve

HideCurve *curve_id*

Hides specified curve

curve_id Id of curve to hide. Curve id is returned by the functions **GetSelectedCurves, GetAxisCurves** and **GetAllCurves**

See Also

ShowCurve

HighlightCurve

HighlightCurve [/clear | /unique] *curveId*

Highlights the selected curve. A curve is highlighted by displaying it in a brighter colour and bringing it to the top - i.e. it is drawn last. Also, highlighted curves are displayed in increased thickness, the amount determined by the HighlightIncrement option setting.

curveId Id of curve to be highlighted (or unhighlighted if /clear specified)

/clear The specified curve will be unhighlighted.

/unique The specified curve will be highlighted and all others will be unhighlighted.

Hint

Hint *message*

Displays a message box intended to be used to provide hints to the user. The box contains a check box allowing the user to choose not to receive such hints again.

message Message to be displayed.

KeepGroup

KeepGroup on|off

Switches *keep status* of current group.

Groups generated by the simulator start with their *keep status* set to off. This means that it will automatically be deleted when a certain number (set by the GroupPersistence option) of new groups are created. If the *keep status* is set to on then automatic deletion is disabled. Groups read from a file using OpenGroup start with their *keep status* set to on.

Let

Let [*vector_expression*]

Evaluates a vector expression.

vector_expression vector expression to be evaluated. Information on vector expressions can be found in Expressions.

To be meaningful *vector_expression* must contain the assignment operator '=' .

If *vector_expressio*n is omitted a list of vectors in the current group will be displayed.

Examples

Create a new vector of name power:

```
let power = r1#p*(r1_p-r1_n)
```

Listing

Listing [/file *filename*] [/errors] [/append *filename*]

Displays or outputs to a file a listing of the current netlist.

/file filename	Result is written to file of name filename
/append filename	Result is appended to file of name filename

ListModels

ListModels *filename*

Generates a dictionary of all models and subcircuits currently available to the simulator (i.e. installed with **File**|**Model Library**|**Add/Remove Libraries** see Pulsonix Spice User's manual). Result is written to *filename*. A single line will be produced for each model or subcircuit found containing the device name, its type (npn, jfet, subcircuit etc.) and the filename in which it was found along with the line number.

ListStdButtonDefs

ListStdButtonDefs [*filename*]

Lists the built in toolbar button definitions. These are in the form of the DefButton command used to create the definition.

filename	If specified, the results will written to filename. Otherwise the results will be displayed in the command shell.

ListStdKeys

ListStdKeys *filename*

Writes built in key definitions to *filename*.

ListStdMenu

ListStdMenu *filename*

Writes the definitions for the built in menu system to *filename*

LoadModelIndex

LoadModelIndex

Forces model library indexes to be re-checked and loaded. Model library indexes are initially loaded when Pulsonix Spice starts. If however some additions are made to the library, it will become necessary to reload them. This command performs that action. Note the menu **Model Library**|**Re-build Catalog** calls this command.

The work of reloading indexes is actually performed in the background so this command returns immediately even though the process can take several seconds. If you start a simulation immediately after executing this command, there will be a pause until the reload is complete.

MakeAlias

MakeAlias *variable*

Converts a string variable to an alias.

variable variable to be converted

An alias is a string representing a numeric expression. For more information see Aliases.

MakeCatalog

MakeCatalog *outfile_name main_catalog* [*user_catalog*]

This command builds a catalog file for use by the Pulsonix Schematics ‘Insert Part Using Model’ function. This is normally called "OUT.CAT" and resides in the Pulsonix-Spice application data folder.

outfile_name File name for catalog. This must be OUT.CAT for use with browser

main_catalog Main database of parts. This would usually be ALL.CAT which resides in the Pulsonix Spice support folder.

user_catalog User database of parts. This would usually be called USER.CAT which resides in the Pulsonix-Spice application data folder.

The **MakeCatalog** command is one of the components of the Pulsonix Schematics ‘Insert Parts Using Model’ option. This option requires a catalog file which lists all the models available to the simulator and for each provides the name of a suitable Pulsonix part, a category, pin mapping info if relevant, a device model property (e.g. X for subcircuits, Q for BJT's) and a preferred pathname if there is more than one model of that name. The **MakeCatalog** command builds this catalog using the data files *main_catalog* and *user_catalog* to obtain information about known models. For more information refer to the Pulsonix Spice User's manual.

MakeCollection

MakeCollection [/multiple] *name*

Makes a group collection. See Accessing Simulation Data for details on collections.

/multiple If specified, a *multiple* collection will be created otherwise it will be *single*. This affects the way graphs are plotted when the /coll switch is specified for **Plot** and **Curve**. When a group is *multiple* a separate legend is created for each curve in the collection and they are plotted using different colours. With *single* collections, the curves are all the same colour and a single legend is created for all of them. *Multiple* collections are used by the stepped analyses while *single* collections are used by Monte-Carlo analysis.

name Name of collection.

MakeTree

MakeTree *path*

Creates the specified directory path. Unlike the MD command, MakeTree will create any subdirectories required to make the whole path.

Mcd

Mcd *directory_name*

Makes a directory and sets it as current. (Same as Md followed by Cd)

directory_name Name of directory to be created.

Md

Md *directory_name*

Creates a new directory. Md is similar to the DOS MD and MKDIR commands.

directory_name Name of directory to be created.

MessageBox

MessageBox *text* [*caption*]

Displays message box with text *text* and caption *caption*.

MoveCurve

MoveCurve *curve_id axis_id*

Moves a curve to a new y-axis

curve_id	Id of curve as returned by Curve id is returned by the functions GetSelectedCurves, GetAxisCurves62 and GetAllCurves
axis_id	Axis id as returned by functions GetAllYAxes, GetSelectedYAxis or GetCurveAxis

MoveFile

MoveFile *path-1 path-2*

Moves path-1 to path-2.

NewAxis

NewAxis

Creates a new y-axis. This will be initially empty and selected. See Pulsonix Spice User's manual for more info on multiple y-axes.

NewGraphWindow

NewGraphWindow

Creates a new graph window to which new graphs may be directed.

NewGrid

NewGrid

Creates a new grid. See Pulsoix SpiceUser's manual for more info on axes and grids.

NewPrinterPage

NewPrinterPage

Advances printer to a new page. This may be used for customised or non-interactive printing. See “Non-interactive and Customised Printing”.

NoPaint

NoPaint

This command has no effect unless executed from within a script. It inhibits all updates to graphs until script execution is complete. This is useful when a number of operations are performed on a graph. By calling this command at the start of a script, multiple graph operations can be performed much faster and more smoothly.

OpenGroup

OpenGroup [/text] [/overwrite] [*filename*]

Reads in a data file.

/*text*	If specified, data file is assumed to be in text format. Otherwise the file is input as a Pulsonix binary data file as saved by the SaveGroup command.
/*overwrite*	Forces existing group of the same name to be overwritten. If not specified, the group being read in will be renamed if a group of the same name already exists.
filename	Name of file to be input. If not specified, an open file dialog box will be opened allowing the user to choose from available files.

OpenGroup creates a new Group. If /text is not specified then the name of the group will be that with which it was stored provided the name does not conflict with an existing group. If there is a conflict the name will be modified to be unique unless /*overwrite* is specified in which case the original group will be destroyed. If /*text* is specified then the group will be named textn where *n* is chosen to make the name unique.

OpenPrinter

OpenPrinter [/portrait] [/numcopies *num_copies*] [/index *index*] [/title *title*] [/printer *printer_name*]

Starts a print session. This may be used for customised or non-interactive printing.

/*portrait*	If specified, print will be in portrait orientation, otherwise it will be landscape.
num_copies	Number of copies to print.
printer_name	Specify printer by name. If omitted, printer will be defined by its index (see below) or the application default printer will be used.
index	Printer to use. This can be found from the function GetPrinterInfo If omitted, the application default printer will be used.
title	Title of print job. This is used to identify a print job and will be displayed in the list of current print jobs that can be viewed for each installed printer from control panel. title is not printed on the final document.

OpenRawFile

OpenRawFile [/purge] [/bufsize *buffer_size*] *rawfile* [*datafile*]

Opens a SPICE 3 format ASCII raw file.

/*purge*	If specified, the loaded data group will be treated like a normal simulation group and will be automatically deleted after three runs. Otherwise it will not be deleted unless the user does so explicitly - e.g. by using the Graphs and Data|Delete Data Group... menu.
/*bufsize buffer_size*	Specifies the percentage proportion of installed RAM that is used for buffering the data. See Notes below for more details. Default value is 10 (%).
rawfile	Raw file to open.
datafile	Pulsonix data file to which data is written - see Notes. If omitted, a file will be created in the temporary data directory as specified by the TempDataDir option setting.

Notes

Note that the data file generated by this command can be reloaded at a later time using the OpenGroup command (or menu File|Data|load...). By specifying the *datafile* argument you can choose the name and location of this file which can be useful for archival purposes.

OptionsDialog

```
OptionsDialog
```

Opens the options dialog box. This is the action performed by the menu **File|Options|General...**. All option processing is performed directly by this command.

Pause

```
Pause
```

Pauses current simulation (if any). Note that this command can only be executed by assigning it to a key or menu item with the direct execution option specified (option flag 5). For more information see User Defined Key and Menu Definitions.

A paused simulation can be restarted with the Resume command.

PlaceCursor

```
PlaceCursor [/main x_main y_main] [/datum x_datum y_datum]
```

Positions graph cursors if they are enabled.

/main *x_main y_main*	Location of main measurement cursor. Position is determined by *x_main*. *y_main* is only used for non-monotonic curves (e.g. nyquist plots) where there is more than one y value for a given x value.
/datum *x_datum y_datum*	Location of reference cursor. Position is determined by *x_datum*. *y_datum* is only used for non-monotonic curves (e.g. nyquist plots) where there is more than one y value for a given x value.

Plot

```
Plot    [/xl x_low_limit x_high_limit]
        [/yl y_low_limit y_high_limit]
        [/xunit x_unit_name]
        [/yunit y_unit_name]
        [/xlabel x_label_name]
        [/ylabel y_label_name]
        [/xdelta x_grid_spacing]
        [/ydelta y_grid_spacing]
        [/title graph_title]
        [/xlog]
        [/ylog]
        [/loglog]
        [/new]
        [/select]
        [/coll]
        [/name curve_name]
        [/dig]
        [/new]
        [/newSheet]
        [/autoxlog]
        [/autoxylog]
        y_expression
        [x_expression]
```

Create a new graph. To modify or add curves to an existing graph, use the **Curve** command. If a graph window is already open, a new tabbed sheet will be created within that window for the new graph unless the /new switch is specified.

/xl	Use fixed limit for x-axis. Followed by ...
x_low_limit	Real. Lower limit of x-axis.
x_high_limit	Real. Higher limit of x-axis.
/yl	Use fixed limit for y-axis. Followed by ...
y_low_limit	Real. Lower limit of y-axis.
y_high_limit	Real. Higher limit of y-axis.
/xunit	Specify units for x-axis (Volts, Watts etc.). Followed by ...
x_unit_name	Text string. Unit name. If it contains spaces, the whole string must be enclosed in quotes (""). You should not include an engineering prefix (m, K etc.) **Pulsonix Spice** will add this automatically as appropriate.
/yunit	Specify units for y-axis. Followed by ...
y_unit_name	Text string. Unit name. Other comments as for x unit name.
/xlabel	Specify label for x-axis. Followed by...
x_label_name	Text String. Label name. If it contains spaces, whole string must be enclosed in quotes ("").
/ylabel	Specify label for y-axis. Followed by...
y_label_name	Text String. Label name. If it contains spaces, whole string must be enclosed in quotes ("").
/xdelta	Specify spacing between major grid lines on x-axis. Followed by...
x_grid_spacing	Real. For default spacing use '0'.
/ydelta	Specify spacing between major grid lines on y-axis. Followed by...
y_grid_spacing	Real. For default spacing use '0'.
/title	Specify title of graph. Followed by ...
graph_title	Graph title
/xlog	Use logarithmic x axis
/ylog	Use logarithmic y axis
/loglog	Use logarithmic x and y axes. Same as /xlog /ylog.
/new	If specified, a new graph window will be opened for the graph.
/select	If specified, the new curve will be selected.
/coll	If specified, the operation will be repeated for all groups in the current collection. This is the method by which multiple curves in a Monte Carlo and step analyses are plotted together.
/name *curve_name*	If specified, curve will be named *curve_name*.
/dig	If specified, the curve will be placed on a digital axis
/new	If specified, a new graph window will be opened.
/newSheet	If specified, a new empty graph sheet will be created in the current graph window. No curves will be plotted.
/autoxlog	If specified, the x-axis will be logarithmic if the x-values are logarithmically spaced. See below for details of test for log spacing.
/autoxylog	Same as /autoxlog except that if x-values are logarithmically spaced, both X and Y axes will be logarithmic

y_expression Text string. Expression describing curve to be added to graph.

x_expression Text string. Expression describing x values of curve defined by y expression. Default is reference of *y_expression*.

Curve legends

The name appearing in the graph window will be the text of the expression being plotted. Sometimes some additional text is appended in parentheses. This is the group name current when the curve was plotted. A special feature exists to allow this appended text to be modified. If there is a string variable in the current group named "legend", its text will be used instead. This facility is used by the parameter stepping feature to label the curve with the value of parameter used for a particular curve.

/autoxlog and /autoxylog log test

The x-values are deemed to be logarithmically spaced if the first three values satisfy the following:

1.0000001 > x1*x1/(x0*x2) > 0.9999999

where x0 is first x-value, x1 is the second and x2 is the third. If there are fewer than three points or any of the values is less than or equal to zero, a linear axis will be selected.

PrintGraph

PrintGraph [/interactive] [/margin *left top right bottom*] [/major on|off] [/minor on|off] [/caption *caption*] [/mono] [*dim_left dim_top dim_right dim_bottom*]

Prints the current graph sheet.

left top right bottom Page margins in mm.

/major on|off Specify whether major grid lines should be printed. Default is on.

/minor on|off Specify whether minor grid lines should be printed. Default is on.

caption Caption printed at the bottom of the page.

dim_left, dim_top, dim_right, dim_bottom
Dimensions and position of printed image on page. Values are relative to page size less the specified margins in units equal to 1/1000 of the page width/height. The default is 0 0 1000 1000 which would place the image to fill the entire area within the margins. 0 500 1000 1000 would place the image at the bottom half of the page. 0 0 2000 1000 would place the left half of the image in the full page while -1000 0 1000 1000 would place the right half. This allows the printing on multiple sheets. Note that if values greater than 1000 or less than 0 are used, part of the printed image will lie in the margins. This provides a convenient overlap for multiple sheets.

/mono If specified, the graph will be printed in black and white

Quit

Quit

Terminates **Pulsonix Spice.**

Rd

Rd *directory_name*

Remove a directory. Rd is similar to the DOS RD and RMDIR commands.

directory_name Name of directory to be removed.

ReadLogicCompatibility

ReadLogicCompatibility *filename*

Reads a file to define the compatibility relationship between logic families. For an example of a compatibility table, see the file COMPAT.TXT which you will find in the SCRIPT directory. This file is actually identical to the built-in definitions except for the "UNIV" family which cannot be redefined.

Please refer to the Pulsonix Spice User's manual for full details on logic compatibility tables.

File format

The file format consists of the following sections:

Header

In-Out resolution table

In-In resolution table

Out-Out resolution table

Header:

The names of all the logic families listed in one line. The names must not use the underscore ('_') character.

In-Out resolution table:

A table with the number of rows and columns equal to the number of logic families listed in the header. The columns represent outputs and the rows inputs. The entry in the table specifies the compatibility between the output and the input when connected to each other. The entry may be one of three values:

OK	Fully compatible
WARN	Not compatible but would usually function. Warn user but allow simulation to continue.
ERR	Not compatible and would never function. Abort simulation.

In-In resolution table

A table with the number of rows and columns equal to the number of logic families listed in the header. Both column and rows represent inputs. The table defines how inputs from different families are treated when they are connected. The entry may be one of four values:

ROW	Row take precedence
COL	Column takes precedence
OK	Doesn't matter. (Currently identical to ROW)
ERR	Incompatible, inputs cannot be connected.

Out-out resolution table

A table with the number of rows and columns equal to the number of logic families listed in the header. Both column and rows represent outputs. The table defines how outputs from different families are treated when they are connected. The entry may be one of four values:

ROW	Row take precedence
COL	Column takes precedence
OK	Doesn't matter. (Currently identical to ROW)
ERR	Incompatible, outputs cannot be connected.

Redirect

Redirect /err | /out [*filename*]

Redirects messages (i.e. text which is normally displayed in the message window) to a file.

Filename Name of file to which messages are sent. If not specified messages are sent to the message window.

One or both of /err or /out must be specified:

/err Specifies that error and warning messages are to be redirected.

/out Specifies that messages other than errors and warnings are to be redirected.

RegisterDevice

RegisterDevice *DLL_name*

This is used to register additional simulator devices defined in an external DLL.

RegisterUserFunction

RegisterUserFunction *Function-Name Script-Name* [*min-number-args*] [*max-number-args*]

Creates a user defined function based on a script.

Function-Name Name of function. This must start with a letter and contain only letters, digits and underscores. The name must not be one of the built-in functions.

Script-Name Name of script that will be called to execute function.

min-number-args Minimum number of arguments required by the function. Range 0 - 7. Default=0

max-number-args Maximum number of arguments that may be supplied to the function. Range 0 - 7. Default=7

Notes

When an expression is evaluated that calls the function defined by this command, the specified script will be called. The script receives the arguments to the function through its argument numbers 2-8. (There is a maximum limit of seven arguments). The function's returned value is the script's first argument passed by reference.

RenameLibs

RenameLibs [/report] [/check] [/log *logfile*] *filename suffix* [*catalog-file user-catalog-file*]

Runs the rename model utility. This renames models inside installed model files if they are found to have duplicates. This command is called by the rename_libs script which is documented in the User's Manual.

filename Name of model library file or file spec to be processed. This may include '*' or '?' wild card characters. Any models within this file that have duplicates already installed in the global model library will be renamed using the suffix supplied.

suffix Suffix applied to duplicate model name.

catalog-file Usually called OUT.CAT. If specified alongside *user-catalog-file,* any user association of renamed models will be appropriately modified.

user-catalog-file Usually called USER.CAT. See catalog-file above.

/report If specified a report of progress will be displayed in the command shell.

/check	If specified a dummy renaming process will be performed. All reports, logs and messages will be output but no actual renaming will take place
logfile	If specified, all renamed models will be listed in logfile.

RepeatLastMenu

RepeatLastMenu *menuname top-menu-name*

Executes the menu most recently selected by the user. The simulator remembers the last command executed for each top level menu and this menu must be specified with this command.

menuname	Identifies the window type that owns the menu. See DefMenu command for list of possible values.
top-menu-name	The top level menu name. For a fixed menu, this is the name that appears in the menu bar. For popup menus, the name "$popup$" must be supplied.

Reset

```
Reset
```

Frees memory associated with most recent simulation run.

It is not normally necessary to use this command unless available memory is low and is needed for plotting graphs or other applications. Note that Reset does not delete the data generated by a simulation only the internal data structures set up to perform a run. These are automatically deleted at the beginning of a new run.

RestartTran

RestartTran *new-tran-stop-time*

Restarts a transient simulation that had previously run to completion. To work, the most recent simulation must have been a transient analysis. If another analysis has since been run or if the analysis has been cleared using the Reset command, this command will be inoperative.

new-tran-stop-time	The restarted run will continue until it reaches this time.

RestDesk

```
RestDesk
```

Restores all windows to positions saved with last SaveDesk command.

Resume

```
Resume
```

Resumes a previously paused simulation.

Run

Run [/check] [/an *analysis_spec*] [/force] [/file] [/nofile] [/options *optionsString*] [/list listFile] [/nolist] [/nodata] [/collection *collection_name*] [/sweep start|continue|finish] [/append *groupname*]

```
[/label div_label] netlist_name [ data_filename ]
```

Runs a simulation on specified netlist.

/check	If specified, simulation is not run but netlist is read in and all checks are performed.

/an *analysis_spec*	If specified, any analysis controls (e.g .TRAN, .AC etc.) in the netlist are ignored and the control in *analysis_spec* is executed instead.
/*force*	*data_filename* will be overwritten if it already exists without prompting user. Otherwise a dialog box will be opened allowing user to select a new file if required.
/*file*	Does nothing. In earlier versions (pre 3.1) this had to be specified to force simulation data to be output to a file. This is now the default behaviour. Specify /nofile to force data to be stored in RAM.
/*nofile*	If specified, simulation data is stored in RAM.
/options *optionsString*	Simulator options settings. *optionsString* may be anything that can be placed after a .OPTIONS control. (Must be enclosed in double quotation marks if *optionsString* contains spaces).
/list *listFile*	Overrides default name for list file.
/*nolist*	Inhibits creation of list file.
/*nodata*	Only data explicitly specified by .PRINT or .KEEP controls will be output. Usually all top level data is saved. Equivalent to placing ".KEEP /nov /noi /nodig" in netlist.
/collection *collection_name*	Attaches group generated by run to collection of name *collection_name*. This is now obsolete and may not be supported in future versions. Collections were used in earlier versions to group data created by multiple runs such as Monte Carlo and stepped analyses. These runs now create a single group containing multi-division vectors.
/*sweep*	May be set to 'start', 'continue' or 'finish'. This is used to create linked runs that save their data to the same group using multi-division vectors. The first run in such a sequence should specify '/sweep start' while the final run should specify '/sweep finish'. All intermediate runs should specify '/sweep continue'. All runs except the first must also specify '/append' - see below.
/append *groupname*	Append data created to specified group name which would always be the data group created by the first run in the sequence. '/sweep continue' or /sweep finish' must also be specified for this to function. The data is appended by adding new divisions to existing vectors so creating or extending a multi- division vector.
/label *division_label*	Used with /sweep to name the division of a linked run.
netlist_name	Input netlist filename.
data_filename	Specifies path name of file to receive simulation data. If omitted, the data is placed in a temporary data file.

Notes

The Run command does not run a simulation on the currently open schematic but on the specified netlist. Normally a run is initiated using the Simulator|Run menu item. This annotates the schematic then generates the netlist using the Netlist command. Run is then executed specifying the new netlist.

The Run command may also be used to run a simulation on a netlist generated by hand or by another schematic editor.

Linking Runs

The data from multiple runs may be linked together in the same manner as multi-step runs such as Monte Carlo. This makes it possible to develop customised multi-step runs using the script language. Simple multi-step runs may be defined using the simulator's built in

features which cover a wide range of applications. The simulator's multi-step features allow the stepping of a single component or a parameter which can define several components. But it doesn't allow, for example, a complete model to be changed, or any kind of topological changes.

The script language may be used to control multiple runs of a circuit with no limit as to the changes that may be performed between each run. In such situations it is useful to be able to organise the data in the same way that the native multi-step facilities use. This can be done by linking runs using the /sweep, /append and /label switches. By running simulations in this manner, the data generated by the simulator will be organised using multi-division vectors which are similar to 2 dimensional arrays.

Care must be taken when making topological changes between runs. Names of nodes that are of interest must always be preserved otherwise the data generated for their voltage may be lost of mixed up with other nodes. Note also that the data for new nodes created since the first run will not be available. The same problems arise for device pin currents.

Linked Run Example

** First run

```
Run /sweep start /label "Run=1" netlist.net

** save group name

Let grp1 = (Groups())[0]



... changes to netlist

** second run

Run /sweep continue /label "Run=2" /append {grp1} netlist.net



... changes to netlist

** third run

Run /sweep continue /label "Run=3" /append {grp1} netlist.net



... changes to netlist

** fourth and final run

Run /sweep finish /label "Run=4" /append {grp1} netlist.net
```

SaveDesk

SaveDesk

Saves the current window positions.. They can be restored using **RestDesk**. (or menu **File|Windows|Restore Desktop**)

SaveGraph

SaveGraph *filename*

Saves the currently selected graph to a binary file. This can subsequently be restored using OpenGraph ()

filename Path of file.

SaveGroup

SaveGroup [/force] [*filename*]

Saves the current group to the *filename* in binary format. Data can later be restored with the OpenGroup command. If *filename* is not specified a dialog box will be opened allowing the user to choose from available files. If /force is specified, any existing file will be overwritten without prompting.

SaveRhs

SaveRhs [/nodeset] *filename*

Creates a file containing every node voltage, inductor current and voltage source current calculated at the most recent analysis point. The values generated can be read back in as nodesets to initialise the dc operating point solution. There are a number of applications for this command - see below.

/*nodeset*	If specified the values are output in the form of a .nodeset command which can be read back in directly. Only node voltages are output if this switch is specified. Otherwise, currents in voltage sources and inductors are also output.
Filename	File where output is written.

This command is intended as an aid to DC operating point convergence. Sometimes the dc operating point solution is known from a previous run but took a long time to calculate. By applying the known solution voltages as nodesets prior to the operating point solution, the new DC bias point will be found much more rapidly. The method is tolerant of minor changes to the circuit. The old solution may not be exact, but if it is close this may be sufficient for the new solution to be found quickly.

This command also has another application for circuits where the DC operating point fails altogether. It is nearly always possible to find the DC bias point using pseudo transient analysis. With this method a transient analysis is performed with the bias point skipped and all voltage and current sources set initially to zero. The sources are then ramped to their initial values much as they would be in a real circuit when the power supplies are switched on. The transient analysis continues until all voltages and currents have settled down. The final result is the dc bias point solution and can be captured using SaveRhs. This can then be applied to the main analysis.

The above procedure is described in more detail in the section on convergence, in the Pulsonix Spice User's manual

If SaveRhs is executed after an AC analysis, the values output will be the real part only.

ScriptAbort

ScriptAbort

Aborts execution of script. Note that this command can only be usefully executed from a key or menu item which has been defined with the direct execution option specified (option flag 5 or /immediate switch for DefMenu). See User Defined Key and Menu Definitions.

See Also

ScriptStep

ScriptResume

ScriptPause

ScriptPause

ScriptPause

Pauses a script. Execution can later be resumed with ScriptResume or single stepped with ScriptStep. Note that this command is often executed from a key or menu item which has been defined with the direct execution option specified (option flag 5 or /immediate for DefMenu). ScriptPause is assigned to shift-F2 by default. Note that it is not possible to use the normal user interface while a script is paused. The main use of script pause is to allow single-stepping for debug purposes.

Scripts can be single stepped by executing ScriptPause immediately before starting the script. If the "EchoOn" option is also enabled, each line of the script as it is executed will be displayed in the message window. See Debugging Scripts.

See Also

ScriptStep

ScriptResume

ScriptAbort

ScriptResume

ScriptResume

Resumes script that has been paused with ScriptPause.

See Also

ScriptStep

ScriptPause

ScriptAbort

ScriptStep

ScriptStep

Steps a paused script by one command. See Debugging Scripts.

See Also

ScriptPause

ScriptAbort

ScriptResume

SelectCursorMode

SelectCursorMode

Switches graph window into "SelectCursor" mode. If the user left clicks the mouse button the graph measurement cursor is placed on the nearest curve. The same operation on the reference cursor may be performed using the right mouse button. Note that normal mode is resumed as soon as the left or right button has been clicked.

SelectCurve

SelectCurve [/unselect] *curve_id*

OR

SelectCurve [/unselect] /all

Selects/unselects the identified curve or all curves

FORM1

Specified curve is selected or unselected.

curve_id Curve id is returned by the functions **GetSelectedCurves**, **GetAxisCurves** and **GetAllCurves**

FORM2

All curves on currently selected graphs are selected or unselected.

BOTH CASES:

/unselect Curve or curves to be unselected.

SelectGraph

SelectGraph *id*

Switches the graph tabbed sheet to the graph specified by *id*.

id Graph id.

SelectLegends

SelectLegends [/unselect]

Selects or unselects all graph window legends.

/*unselect* If specified, all legends are unselected. Otherwise they are selected.

Set

Set [/temp] [*option_spec* [*option_spec*...]]

Defines an option.

/*temp* If specified, the setting will only remain for the duration of the current script execution. Value will return to its original setting when control returns to the command line.

option_spec Can be one of two forms:
Form1: *option_name*
Form2: *option_name* = *option_value*

option_name Can be any of the names listed in the options section of the "Sundry Topics Chapter" of the *User's Manual*. For options of type Boolean, use form1. For others, use form 2.

SetCurveName

SetCurveName *curve_id curve_name*

Changes curve name. This is the name displayed in the legend panel.

curve_id Curve Id. Curve id is returned by the functions **GetSelectedCurves**, **GetAxisCurves** and **GetAllCurves**

curve_name New name for curve.

SetGraphAnnoProperty

SetGraphAnnoProperty *object-id property-name property-value*

Sets a property value for a graph object. Note that this command's name is a little misleading as it can edit the values of the properties of any graph object not just *annotation* objects

object-id	Id of object which owns the property to be edited.
property_name	Name of property to be edited
Property-value	New value of property.

SetGroup

SetGroup *group_name*

Changes the current group.

group_name	Name of new group. An array of current group names is returned by the **Groups** function.

SetRef

SetRef *vector_name reference_expression*

Attaches *reference_expression* to *vector_name.* Previous reference is detached and deleted if no longer used.

See Also

Expressions

SetToolBarVisibility

SetToolBarVisibility *toolbar_name visibility*

Sets the visibility of a toolbar.

toolbarname	Either a pre-defined toolbar or a new one.
visibility	Specifies when toolbar is visible.

SetUnits

SetUnits *vector_name physical_type*

Changes physical type of *vector_name* to *physical_type*. Physical type may be any of the following:

"unknown"	"?"
"Voltage"	"V"
"Current"	"A"
"Time"	"Secs"
"Frequency"	"Hertz"
"Resistance"	"Ohm"
"Conductance"	"Sie"
"Capacitance"	"F"
"Inductance"	"H"
"Energy"	"J"
"Power"	"W"
"Charge"	"C"
"Flux"	"Vs"
"Volt^2"	"V^2"
"Volt^2/Hz"	"V^2/Hz"
"Volt/rtHz"	"V/rtHz"
"Amp^2"	"A^2"
"Amp^2/Hz"	"A^2/Hz"
"Amp/rtHz"	"A/rtHz"

The physical type of a vector is the name of the physical quantity it represents e.g. Voltage, Current, Time etc. This is used by graph plotting routines to set appropriate units for axes.

Shell

Shell [/wait] [/displayStdout] [/command *command_string*] *application_name*

Launches an application.

/wait	If specified, application is launched synchronously. This means that the simulator will not continue until the application has closed.
/displayStdout	Displays in the message window any standard output from the program
/command *command_string*	Calls system command processor to execute *command_string*. This is necessary to run internal commands such as Copy and Move. The command processor is usually CMD.EXE
application_name	File system path to executable file. This would usually be a binary executable but may be any file that is defined as executable by the operating system. If a full path is not specified, a search will be made for the file.

Notes

To run a console mode application in a manner such that the console is displayed, use the ShellOld command (see below).

ShellOld

Shell [/wait] [/icon|hide] [/command *command_string*] *application_name*

Launches an application

/wait	If specified, application is launched synchronously. This means that the simulator will not continue until the application has closed.
/command *command_string*	Calls system command processor to execute *command_string*. This is necessary to run internal commands such as Copy and Move. The command processor is usually CMD.EXE
/icon|hide	If /icon, the program is started in a minimised state. If /hide, the program main window is initially hidden.
application_name	File system path to executable file. This would usually be a binary executable but may be any file that is defined as executable by the operating system. If a full path is not specified, a search will be made for the file.

Show

Show [/file *filename*] [/append *filename*] [/noindex] [/noHeader] [/plain] [/force] [/names *names*] [/clipboard] [/width *width*] [/lock] *expression* [*expression* ...]

Displays the value of an expression.

/file *filename*	If specified, outputs result to filename. The values are output in a format compatible with OpenGroup /text.
/append *filename*	As /file except that file is appended if it already exists.
/noindex	If the vector has no reference, the index value for each element is output if this switch is *not* specified.
/noHeader	If specified, the header providing vector names etc. will be inhibited.
/plain	If specified, no index (as /noindex), and no header (as /noHeader) will be output. In addition, string values will be output less the quotation marks.

/force	File specified by /file will be unconditionally overwritten if it exists.
/names *names*	Semicolon delimited list of column labels. If specified, each vector column will be labelled by the corresponding name given in *names*. Otherwise, vector name is used as label.
/clipboard	If specified, the result is copied to the windows clipboard.
/width	Page width in columns.
/lock	If specified with /file, a lock file will be created while the write operation is being performed. The file will have the extension .lck. This can be used to synchronise data transfers with other applications. The file will be locked for write operations
expression	Expression to be displayed. If *expression* is an array, all values will be displayed.

Notes

To enter multiple expressions, separate each with a comma.

The display of arrays with a very large number of elements (>500) can take a long time. For large arrays it is recommended that the /file or /clipboard switch is used to output the results to a file or the windows clipboard respectively. The data can then be examined with a text editor or spreadsheet program.

The results will be tabulated if all vectors are compatible that is have the same x- values. If the any vectors listed are not compatible, each vector's data will be listed separately.

The precision of numeric values can be controlled using the “Precision” option setting. Use the command “Set precision = *value*”. This sets the precision in terms of the column width.

ShowCurve

ShowCurve

Shows specified curve having been hidden using HideCurve

curve_id	Id of curve to show. Curve id is returned by the functions **GetSelectedCurves, GetAxisCurves** and **GetAllCurves**

See Also

HideCurve

ShowSimulatorWindow

ShowSimulatorWindow

Displays simulator status window if it is currently hidden.

SizeGraph

SizeGraph [/xfull] [/yfull] [/axisid *axis_id*] *xoffset yoffset xscale yscale*

General purpose command to zoom or scroll a graph.

/xfull	If specified, the x-axis is zoomed to fit whole graph. *xscale* and *xoffset* will be ignored
/yfull	If specified, the y-axis is zoomed to fit whole graph. y*scale* and y*offset* will be ignored
/axisid *axis_id*	Specify which y-axis to resize. If omitted, all y-axes on selected graph will be affected.
xoffset	Extent of X-shift as proportion of full width of graph. E.g. 0.25 will shift by a quarter. 0 has no effect.

yoffset	As *xoffset* but for y-axis
xscale	View width required as proportion to current width. E.g. 0.8 will zoom in by 20%. 1 has no effect. 0 is illegal.
yscale	As *xscale* but for y-axis.

Stats

Stats

Displays statistics relating to most recent simulation.

Note that time values other than total analysis time will not be supplied unless the simulator option “TIMESTATS” or “ACCT” is specified.

Title

Title graph *new_title*

Changes a window's title.

graph	Apply to selected graph window
new_title	New window title

Notes

The title is displayed in the window's caption bar and is also placed at the bottom of printed graphs.

Trace

Trace *signal_name trace_id*

The trace command is used to set up a simulation trace while a simulation is running. To set up a trace before a simulation is started, use the .TRACE or .GRAPH simulator controls.

signal_name	Net name or pin name for voltage or current to be traced.
trace_id	Integer value used to group traces together on the same graph. All traces with the same *trace_id* will go to the same graph.

Note that traces set up with this command only remain in effect until the end of the simulation. A Trace command executed before a simulation starts will have no effect.

UndoGraphZoom

UndoGraphZoom

Restores previous graph view area. Successive execution of this command will retrace the entire history of graph magnification and scroll positions.

UnHighlightCurves

UnHighlightCurves

Unhighlights all curves.

Unlet

Unlet *vector_name*

Destroy vector.

vector_name Name of vector to be destroyed. Unless the vector is in the *user* group, the vector's full qualified name must be used.

See Also

Expressions

Let

Unset

UnSet [/temp] *option_name*

Deletes specified option.

/temp Deletes only temporarily. Will revert to original value once control returns to the command line.

Note that some Option values are internal. This means that they always have a value. If such an option is UnSet, it will be restored to its default value and not deleted.

ViewFile

ViewFile *filename*

Opens a read only file viewer with specified file name. The file viewer is internal while the file editor called by **EditFile** is an external program.

Wait

Wait

Suspends command execution until any mouse key is clicked. Wait does not suspend commands executed directly on assignment to keystrokes or menu items. This allows the cancel command, when assigned to a key or menu, to terminate a wait command.

Where

This command has been deleted. Instead an internal script of the same name exists, which performs the same task as the original Where command.

The Where script displays convergence information about the most recent run.

WriteImportedModels

WriteImportedModels *netlist filename*

Writes all library models required by *netlist* to *filename*

Chapter 4. Applications

User Interface

A full description of the user interface is outside the scope of this manual. Instead, in this section, we provide a few pointers on how to go about finding how a particular feature works so that it can be altered or adapted.

User Defined Key and Menu Definitions

Virtually the entire user interface is accessed through menus or keys all of which may be redefined, deleted or replaced. The only parts of the UI which are not accessible are the tool bar buttons and the mouse keys. These have fixed definitions and may not be modified by the user.

In principle it is possible to define completely new menus which bear no similarity with the built-in menus. A more normal use of menu and key redefinition would probably be to add a special function or perhaps to delete some unused menus.

Menus are defined using the **DefMenu** command and keys can be defined with the **DefKey** command. Unless you only wish to make a temporary definition it is best to place **DefMenu** and **DefKey** commands in the start up script so that they are automatically defined for all future sessions.

Key definitions may be *context sensitive*. That is, the definition is dependent on which type of window is currently active. See **DefKey** command for more details.

To find out the standard definition for menus and keys, use the commands **ListStdMenu** and **ListStdKeys** respectively. These list such definitions to a file. A large number of these simply call a script that performs the menu function. Others carry out the relevant operation directly.

Rearranging or Renaming the Standard Menus

The standard menu definitions are loaded from the built in script ‘menu’ when the program first starts. The source for all built in (or internal) scripts can be found on the install CD. To modify any of the standard menus, you need to modify the ‘menu’ script. For details on how to modify internal scripts.

When editing menu.sxscr, please note the following:

- Each menu definition must occupy a single line
- Menus are created in the order they appear in the script. To change the order, simply rearrange the lines.
- You can disable a menu definition by putting a '*' as the first character of the line. This makes it easy to later undelete it.

Menu Shortcuts

These are keys which activate defined menus. The key name is displayed to the right of the menu text. All menu definitions may have shortcuts specified using the "/shortcut" switch for the **DefMenu** command. A potential problem arises if the same key is used for a shortcut and a key definition using **DefKey**. If this happens, the **DefKey** definition takes precedence.

Modifying Internal Scripts

The Pulsonix-Spice user interface is implemented with about 300 internal (or built-in) scripts. These are built in to the executable file but can be bypassed so that their function can be changed. The code for most of these scripts can be found on the installation CD in directory scripts. The procedure for replacing an internal script is very straightforward. Simply place a script with the same name but with the extension .sxscr in the built-in script directory. The location of this directory is set in the file locations sheet of the options dialog box (menu File|Options|General...). This is usually <Pulsonix root>\Pulsonix-Spice\support\biscript. Pulsonix-Spice always searches this directory first when executing an internal script.

Custom Curve/Performance/Histogram Analysis

The graph menu **Measure|More Functions** opens a tree list dialog box that displays the functions available. In this section we describe how this system works and how it can be extended.

We have only skimmed over the basics. For more information, please refer to the scripts themselves.

Adding New Functions

The measure operations listed for the graph menu described above are obtained from a built-in text file "analysis_tree.sxscr".

Like built in scripts, this is embedded in the binary executable but can also be overridden by placing a file of the same name in the biscript directory.

Each entry in the tree list is defined by a single line in the file. Each line contains a number of fields separated by semi-colons. The first field is that command that is called to perform the action while the remaining fields describe the hierarchy for the entry in the tree list control. The command is usually a script often with a number of arguments. To add a new function, simply add a new line to the file. The order is not important.

"measure", "measure_span", "performance" and "mc_histo" Scripts

These are the "driver" scripts that perform the curve analysis, curve analysis over cursor span, performance analysis and histogram analysis respectively. These don't perform the actual calculations but carry out a number of housekeeping tasks. The calculations themselves are performed by a script whose name is passed as an argument. To add a new function you need to create one of these scripts. For simple functions the script is not complicated. In the example below we show how the "Mean" function is implemented and you will see that it is very simple.

An Example: The "Mean" Function

The entry for the "Full" version of this in analysis_tree.txt is:

```
measure_mean;Measure;Transient;Mean;Full
```

This means that the script `measure_mean` will be called when this function is selected. `measure_mean` is quite simple, it is just a single line

```
measure /ne 'calculate_mean' 'Mean'
```

`/ne` is not that important, it just tells the script system not to enter the command in the history list.

`'calculate_mean'` specifies the script to call to perform the calculation.

`'Mean'` specifies the y-axis label

The `calculate_mean` script is as follows:

```
Arguments data xLower xUpper @result @error




if xUpper>xLower then
    Let result = Mean1(data, xLower, xUpper)
else
    Let result = Mean1(data)
endif
```

The argument `data` is the data that is to be processed. In this case we simply need to find its Mean. `xUpper` and `xLower` specify the range over which the mean should be calculated. These would be specified if the “cursor span” version of the mean function was selected by the user. The result of the calculation is assigned to the argument `result` which has been "passed by

reference". The `error` argument is not used here but it can be used to signal an error condition which will abort the operation. This is done by setting it to 1.

Automating Simulations

Overview

The script language allows you to automate simulations, that is automatically run a number of simulation runs with different component values, test conditions or analysis modes. This section describes the various commands needed to do this.

Running the Simulator

Simulations are started using the Run command. The Run command runs a netlist not a schematic, so you must first create the netlist using the NetList command. Some notes about the Run command:

1. The /an switch is very useful and allows you to run different analyses on the same circuit without having to modify it. /an specifies the analysis mode instead and overrides any analysis controls (e.g. .TRAN, .DC etc.) in the circuit itself.
2. If the run fails (e.g. due to non-convergence), the script will abort without performing any remaining runs. This behaviour can, however, be inhibited with the /noerr switch which must be placed immediately after the Run word:

   ```
   Run /noerr /file design.net
   ```

   /noerr is a general switch that can be applied to any command. If you want to test whether or a run was successful, use the GetLastError function.

Changing Component Values or Test Conditions

It is likely that in an automated run you will want to change component values or stimulus sources between runs. There are a number of ways of doing this, each with its own advantages and disadvantages.

Circuit Parameters

Specify the component values as parameters in the Schematic and then vary the parameter using the **Let** command. To do this, you must first use Pulsonix Schematics to edit the value of the components to be varied so that they are represented as a parameter expression enclosed by curly braces '{' and '}'. Again we will use the example of a resistor R5 whose value we wish to set to 12K. Proceed as follows:

1. In the schematic, select R5 then press shift-F7. Enter {R5} as the new value.
2. Now in the script you can set the value of R5 with **Let** e.g.

   ```
   Let global:R5=12k
   ```

   The "global:" prefix is necessary to make the parameter global. Note we have named the parameter "R5". This is an obvious choice of parameter name but you could use anything as long as it starts with a letter and consists of letters numbers and the underscore character. (You *can* use other characters but we don't recommend it).

You can use the same technique for model names and stimulus specifications. Note that a model name is a string and has to be enclosed in single quotation marks when assigned with a **Let** command. E.g.

```
Let global:Q1='Q2N2222'
```

(The different uses of single and double quotation marks causes confusion. This is explained in Quotes: Single and Double.

For stimulus specifications there are two methods. You can either use a parameter to vary a single value such as the pulse width or you can use a string parameter to vary the entire specification. For a single value - say the pulse width - set the schematic component value (using shift-F7) to "Pulse 0 5 0 10n 10n {PW} 2.5u". Then in the script the pulse width can be changed with:

```
Let global:PW=1u
```

To vary the whole specification, set the component value to - say - "{PULSE_SPEC}" then in the script:

```
Let global:PULSE_SPEC = 'Pulse 0 5 0 10n 10n 1u 2.5u'
```

An alternative, and somewhat more sophisticated approach is to change the component value to parameter version (e.g. "{R5}") in the netlist itself. You could then run simulation on the netlist with parameterised values after which the components in the netlist can be restored to their original values. That way the netlist is preserved with its original values. To do this correctly you would need to save the original values so that they can be restored. The example shown below uses this technique.

Multiple Netlists

Conceptually this is probably the simplest approach but not very flexible. Simply create multiple versions of the netlist manually with different file names then run them one at a time.

An Advanced Example - Reading Values from a File

In this section we supply an example of quite an advanced script which runs several simulations on a netlist. On each run a number of components are changed to values read in from a file. This script is general purpose and can be used for any netlist without modification.

The script is quite complicated but is well commented throughout to explain in detail how it works. The basic sequence is as follows:

1. Get netlist file name from user
2. Get definition file name from user
3. Read first line of the definition file. This has the names of the components to be modified
4. Temporarily edit the specified components' values in the netlist to reference a parameter, saving the original values.
5. Read the rest of the definition file and write the values for each run to an array
6. Run the simulations
7. Restore original values in the netlist
8. Clean up before exit

Here is the script. It is also supplied on the disk in the scripts directory in a file called ‘multiple_simulations.sxscr’.

```
**-------------------------------------------------------------
** Get the netlist from the user
**-------------------------------------------------------------


** First see if there is a current netlist, and use it if there is
Let netlist = global:lastNetlist


if Length(netlist)=0 then
    Let netlist = GetUserFile("'Netlist Files|*.net;*.cir'", 'net', ['Open'])


    if Length(netlist)=0 then
        ** cancelled
        exit script
```

```
    endif
endif

Let netlistLines = ReadFile(netlist)
Let numNetlistLines = Length(netlistLines)

** Test if enough lines

if numNetlistLines<2 then
    Echo "Invalid netlist file"
    exit script
endif

**-------------------------------------------------------------
** Get the parameter definition file from the user
**-------------------------------------------------------------

Let filename = GetUserFile("'Parameter Definition Files|*.txt'", 'txt', ['Open'])

if Length(filename)=0 then
    ** cancelled
    exit script
endif

Let lines = ReadFile(filename)

Let numLines = Length(lines)

** Test if enough lines

if numLines<2 then
    Echo "Definition file must have at least two lines"
    exit script
endif

**-------------------------------------------------------------
** Read in the component names
**-------------------------------------------------------------

Let components=Parse(lines[0])
Let numComponents = Length(components)

if numComponents=0 then
    Echo "No component names specified on first line of definition file"
    exit script
endif
**-------------------------------------------------------------
** Change the netlists component values to parameters
**-------------------------------------------------------------

Let originalNetlistLines = MakeString(numNetlistLines)
Let originalNetlistLines = netlistLines
```

```
** scan through list of components
for compIndex =0 to numComponents-1
    Let found = 0;

    for netlistIndex=1 to numNetlistLines-1

        ** Parse the line into individual values
        Let netlistValues = Parse(netlistLines[netlistIndex])
        Let numNetlistValues = Length(netlistValues)

        if numNetlistValues>0  then
            ** We have a device name, unless it starts with a dot
            ** Use numNodes = 999 to signify an invalid line
            Let numNodes = 999

            Let firstChar = Char(netlistValues, 0)
            if firstChar = 'B' then
                Let numNodes = 2
            elseif firstChar = 'C' then
                Let numNodes = 2
            elseif firstChar = 'D' then
                Let numNodes = 2
            elseif firstChar = 'E' then
                Let numNodes = 4
            elseif firstChar = 'F' then
                Let numNodes = 2
            elseif firstChar = 'G' then
                Let numNodes = 4
            elseif firstChar = 'H' then
                Let numNodes = 2
            elseif firstChar = 'I' then
                Let numNodes = 2
            elseif firstChar = 'J' then
                Let numNodes = 3
            elseif firstChar = 'M' then
                Let numNodes = 4
```

```
            elseif firstChar = 'O' then
                Let numNodes = 4
            elseif firstChar = 'Q' then
                Let numNodes = 3
            elseif firstChar = 'R' then
                Let numNodes = 2
            elseif firstChar = 'S' then
                Let numNodes = 4
            elseif firstChar = 'T' then
                Let numNodes = 4
            elseif firstChar = 'V' then
                Let numNodes = 2
            elseif firstChar = 'Z' then
                Let numNodes = 3
            endif
```

```
            if numNetlistValues>numNodes+1 then
            ** got enough entries (name ,  nodes and a value)

                if netlistValues[0]={components[compIndex]} then
                    ** found component
                    ** Set new value as a parameter with same name as comp reference
                    Let netlistValues[3] = "'{' &  components[compIndex] & '}'"

                    ** re-create netlist line with parameter in it
                    Let netlistLines[netlistIndex] = netlistValues[0]

                    for valIndex=1 to numNetlistValues-1
                        Let netlistLines[netlistIndex] = netlistLines[netlistIndex] & ' ' &
netlistValues[valIndex]
                    next valIndex

                    Let found = 1
                    exit for
                endif
            endif
        endif
    next netlistIndex
    if found=0 then
        Echo "Cannot find component " {components[compIndex]}
    endif

next compIndex

Show /plain /force /file {netlist} netlistLines

**-------------------------------------------------------------
** Read in the component values
**-------------------------------------------------------------

Let compValues = MakeString(numComponents*(numLines-1))
Let error = 0
Let resIndex=0

for lineIndex=1 to numLines-1
    ** Parse the line into individual values
    Let values = Parse(lines[lineIndex])

    if Length(values)<>numComponents then
        ** A line found with the wrong number of values. This is assumed
        ** to be a mistake unless the line is completely empty
        if Length(values)<>0 then
            Echo "Wrong number of values at line " {lineIndex}
            Let error = 1
        endif
    else
        ** line is OK so write values to compValues
        for index=0 to numComponents-1
```

```
            Let compValues[resIndex*numComponents+index] = values[index]
        next index

        Let resIndex = resIndex+1
    endif
next lineIndex

if error then
    exit script
endif

** resIndex finishes with the number of non-blank data lines
Let numRuns = resIndex

**-------------------------------------------------------------
** run the circuit
**-------------------------------------------------------------

for index=0 to numRuns-1
    for compIndex=0 to numComponents-1
        Let paramName = 'global:' & components[compIndex]
        Let {paramName} = compvalues[index*numComponents+compIndex]
    next compIndex;

    Run /file {netlist}
next index

**-------------------------------------------------------------
** restore netlist to original
**-------------------------------------------------------------

Show /plain /force /file {netlist} originalNetlistLines

**-------------------------------------------------------------
** delete global varaibles
**-------------------------------------------------------------

for compIndex=0  to numComponents-1
    Let paramName = 'global:' & components[compIndex]
    UnLet {paramName}
next compIndex

**-------------------------------------------------------------

exit script
```

Data Import and Export

This section is also in the Pulsonix SpiceUser's manual. It is reproduced here for convenience.

Pulsonix Spice provides the capability to export simulation data to a file in text form and also to import data from a file in text form. This makes it possible to process simulation data using another application such as a spreadsheet or custom program.

Importing Data

To import data use the OpenGroup command with the /text switch. E.g. at the command line type:

```
OpenGroup /text data.txt
```

This will read in the file data.txt and create a new group called text*n*. See Data Files Text Format below for details of format.

Exporting Data

To export data, use the **Show** command with the /file switch. E.g

```
Show /file data.txt vout r1_p q1#c
```

will output to data.txt the vectors vout, r1_p, and q1#c. The values will be output in a form compatible **OpenGroup** /text.

Launching Other Applications

Data import and export makes it possible to process simulation data using other applications. **Pulsonix Spice** has a facility to launch other programs using the **Shell** command. You could therefore write a script to export data, process it with your own program then read the processed data back in for graphing. To do this you must specify the /wait switch for the Shell command to force **Pulsonix Spice** to wait until the external application has finished. E.g.

```
Shell /wait procdata.exe
```

will launch the program procdata.exe and will not return until procdata.exe has closed.

Data Files Text Format

There are two alternative formats.

The first is simply a series of values separated by whitespace. This will be read in as a single vector with a reference equal to its index.

The second format is as follows:

A text data file may contain any number of *blocks*. Each *block* has a *header* followed by a list of *datapoints*. The *header* and each *datapoint* must be on one line.

The *header* is of the form:

reference_name *ydata1_name* [*ydata2_name* ...]

Each *datapoint* must be of the form:

reference_value *ydata1_value* [*ydata2_value* ...]

The number of entries in each *datapoint* must correspond to the number of entries in the *header*.

The *reference* is the x data (e.g. time or frequency).

Example

Time	Voltage1	Voltage2
0	14.5396	14.6916
1e-09	14.5397	14.6917
2e-09	14.5398	14.6917

```
4e-09                14.54                14.6917
8e-09                14.5408              14.6911
1.6e-08              14.5439              14.688
3.2e-08              14.5555              14.6766
6.4e-08              14.5909              14.641
1e-07                14.6404              14.5905
1.064e-07            14.6483              14.5821
```

If the above was read in as a text file (using OpenGroup /text), a new group called textn where n is a number would be generated. The group would contain three vectors called time, “Voltage1” and “Voltage2”. The vectors “Voltage1” and “Voltage2” would have a *reference* of “Time”. “Time” itself would not have a *reference*.

To read in complex values, enclose the real and imaginary parts in parentheses and separate with a comma. E.g:

```
Frequency            :               VOUT
1000                                 (-5.94260997 ,0.002837811 )
1004.61579                           (-5.94260997 ,0.00285091  )
1009.252886                          (-5.94260996 ,0.002864069 )
1013.911386                          (-5.94260995 ,0.002877289 )
1018.591388                          (-5.94260994 ,0.00289057  )
1023.292992                          (-5.94260993 ,0.002903912 )
1028.016298                          (-5.94260992 ,0.002917316 )
1032.761406                          (-5.94260991 ,0.002930782 )
1037.528416                          (-5.9426099  ,0.00294431  )
1042.317429                          (-5.94260989 ,0.0029579   )
1047.128548                          (-5.94260988 ,0.002971553 )
1051.961874                          (-5.94260987 ,0.002985269 )
```

Graph Objects

Overview

Graph objects are the items displayed in a graph window. These include curves, axes, cursors and the various objects used for annotation. All graph objects possess a number of named properties all of which may be read and some may also be written. Each graph object also has a unique id which is used to identify it.

A knowledge of the inner workings of graph objects will be useful if you wish to customise some of the annotation features provided by the waveform viewer. However, the interface is at a low level with much work carried out by internal scripts. Consequently there is quite a steep learning curve to climb in order to make good use of the features available.

Object Types

The following table lists all the available object types

Object name	Description
Axis	Axes and grids
Crosshair	Crosshair part of cursor
CrosshairDimension	Object used to dimension cursors. Forms part of cursor. Cannot be displayed on its own
Curve	Curve
CurveMarker	Marker used to annotate curves
FreeText	Free Text annotation object. Displays unboxed text on graph
Graph	Graph sheet
LegendBox	Box enclosing LegendText objects
LegendText	Text objects used in legend boxes and linked to a displayed curve.
TextBox	Box enclosing FreeText object

Properties

Properties are the most important aspect of graph objects. Each type of graph object possesses a number of properties which determine characteristics of the object. Some properties are read only and are either never altered or can only be altered indirectly. Other properties can be changed directly using the command SetGraphAnnoProperty. The labels for curves, axes and the various annotation objects are examples of properties that may be edited.

Graph Object Identifiers - the “ID”

Each instance of a graph object is uniquely identified by an integer value known as its “ID”. Valid IDs always have a value of 1 or greater. IDs are returned by a number of functions (see below) and also a number of the objects possess properties whose value is the ID of a related object.

Once the ID of an object has been obtained, its property names can be read and it property values may be read and/or modified.

The following functions return graph object IDs. Note that all functions return object IDs belonging to the currently selected graph only except for GetGraphObjects which can optionally return IDs for objects on a specified graph.

GetAllCurves	Returns the IDs for all curves
GetAllYAxes	Returns the IDs for all Y-axes
GetAxisCurves	Returns IDs for all curves attached to specified y- axis
GetCurrentGraph	Returns the ID for the currently selected graph sheet
GetCursorCurve	Returns information about curve attached to the main cursor including its ID
GetCurveAxus	Returns ID of y-axis associated with a curve.
GetDatumCurve	Returns information about curve attached to the reference cursor including its ID
GetGraphObjects	Returns all objects on a graph, or objects of a specified type
GetSelectedCurves	Returns IDs of all selected curves
GetSelectedGraphAnno	Returns ID of the selected annotation object
GetSelectedYAxis	Returns the ID of the selected Y-axis
GetXAxis	Returns the ID of the X-axis

Some of the functions in the above list are technically redundant. For example the value obtained by GetCurveAxis() can also be obtained by reading the value of the 'Y-axis' property of the curve. This can be done with the general purpose GetGraphObjPropValue function.

Symbolic Values

Some properties used for labels may be given symbolic values. Symbolic values consist of a property name enclosed with the '%' character. When the label is actually displayed the property name is replaced with its value.

Symbolic values may also be indirect. Some properties return the id of some other associated object and the value of a property for that object may be referenced with a symbolic value. The ':' character is used to denote indirect symbolic values. For example, this method is used with curve markers. The default value for a curve marker's label is:

```
%curve:label%
```

“curve” is a property of a curve marker that returns the id of the curve that it points to. “label” is a property of a curve that returns the label assigned to it. So “curve:label” returns the label of the curve that the curve marker points to.

Other curve properties can be used for this label. For example, curve measurements (as displayed below the legend in the legend panel) can also be accessed via property named “measurements”. So the curve marker label:

```
%curve:label% %curve:measurements%
```

would display the curve's name followed by its measurements.

Finally the character sequence <n> can be used to denote a new line.

Objects and Their Properties

The following lists all the properties available for all objects. Note that all objects have a 'Type' property that resolves to the object's type name. Also all objects except Graph have a 'Graph' property that returns the ID of the object's parent graph sheet.

Axis

Axis objects represent both x and y graph axes

Name	Description	Read Only?
Type	Type of object - always 'Axis'	Yes
Graph	ID of parent graph	Yes
AxisType	'X', 'Y' or 'Dig'	Yes
Label	Label used to annotate axis. (Actual displayed text is <label> / <unit>). Default = %DefaultLabel%	No
Name	Axis name. ('Y1', 'Y2' etc.). Empty for X and Dig axes	Yes
Unit	Physical units of axis. (E.g. 'V', 'A' etc.). Default = %DefaultUnit%	No
Min	Minimum limit of axis	No
Max	Maximum limit of axis	No
AutoLimit	'On' or 'Off' determines whether axis limits are adjusted automatically according to attached curves	No
Grad	Grading of axis: “Log” or “Lin”.	No
Delta	Value that determines the minor grid line spacing	No
DefaultLabel	Label property is given default value of %DefaultLabel% which resolves to the value of this property.	Yes
DefaultUnit	Unit property is given default value of %DefaultUnit% which resolves to the value of this property.	Yes

Crosshair

Object used to display cursor. Each graph cursor consists of a Crosshair and two CrosshairDimensions.

Name	Description	Read Only?
Type	Type of object - always 'Crosshair'	Yes
Graph	ID of parent graph	Yes
X1	X data value of crosshair position	No
Y1	Y data value of crosshair position. The value can be written but this can only affects non-monotonic curves where there are multiple y crossings at a given x value.	No
XDimension	The ID for the CrosshairDimension object that displays the X dimension and positions	Yes
YDimension	The ID for the CrosshairDimension object that displays the Y dimension and positions	Yes
Point	1 = Main cursor. 2 = Reference cursor	Yes
Curve	ID of attached curve	No
Division	Division index of attached curve. See page for details on multi-division vectors	No
Style	Style of crosshair. Possible values: 'Crosshair' or 'Cursor'. 'Crosshair' means the object is displayed as a crosshair with a width and height that extends to cover the whole grid. 'Cursor' means that the object is a small bitmap representing a cross.	No
Frozen	'On' or 'Off'. If 'On' the user will not be able to move the cursor with the mouse	No
OldStepMethod	'On' or 'Off'. Selects method of choosing the position of the cursor when stepped to a new curve using the TAB key.	No

CrosshairDimension

Object used to display the dimensions and positions of cursors. There are two types, namely horizontal and vertical.

Name	Description	Read Only?
Type	Type of object - always 'CrosshairDimension'	Yes
Graph	ID of parent graph	Yes
X1	For horizontal types, holds the value of the x data position of the first crosshair and is readonly. For vertical types holds the x view co- ordinate location of the object on the screen and is writeable	No
Y1	For vertical types, holds the value of the y data position of the first crosshair and is readonly. For horizontal types holds the x view co- ordinate location of the object on the screen and is writeable	No
X2	For horizontal types, holds the value of the x data position of the second crosshair and is readonly. For vertical types holds the view co- ordinate location of the object on the screen and is writeable	No
Y2	For vertical types, holds the value of the y data position of the second crosshair and is readonly. For horizontal types holds the x view co-ordinate location of the object on the screen and is writeable	No
XDiff	= X2-X1	No

YDiff	= Y2-Y1	No
Label1	Label positioned to depict value of first crosshair. Default = %x1% for horizontal types, %y1% for vertical.	No
Label2	Label positioned to depict value of second crosshair. Default = %x2% for horizontal types, %y2% for vertical.	No
Label3	Label positioned to depict the separation between crosshairs. Default = %XDiff% for horizontal types, %YDiff% for vertical.	No
Style	Controls display of dimension labels. Possible values:	No

Internal	Show difference only (label3) - internal position
External	Show difference only (label3) - external position
Auto	Show difference only (label3), position chosen automatically
P2P1	Show absolute labels (label1 and label2)
P2P1Auto	Show all labels. Difference position chosen automatically
None	No controls selected

Font	Font used to display labels. Can either be the name of a font object or a font spec as returned by GetFontSpec.	No
Vertical	0 = Horizontal dimension, 1 = Vertical dimension	Yes
Curve1	ID of curve attached to crosshair 1	Yes
Curve2	ID of curve attached to crosshair 2	Yes
Crosshair1	ID of crosshair 1	Yes
Crosshair2	ID of crosshair 2	Yes

Curve

Curve objects represent all graph curves

Name	Description	Read Only?
Type	Type of object - always 'Curve'	Yes
Label	The curve's label. This is the text that appears in the legend panel. This can use a symbolic constant and in fact defaults to %DefaultLabel%. Note that when reading back a symbolic value assigned to this property, the resolved value will be returned	No
Name	The curve's base name. This is the value passed to the /name switch of the Curve/Plot command or the name of the vector plotted if no /name switch is supplied.	No
Suffix	This is assigned when the result of a multi- step analysis is plotted to give information about the step. E.g. if you stepped a resistor value the suffix would hold the name and value of the resistor at the step.	No
Graph	ID of parent graph	Yes
GroupName	The data group that was current when the curve was created	Yes

DefaultLabel	This is composed from Name, Suffix and GroupName to form a text string that is the default label for the curve	Yes
Measurements	Measurements added to a curve	Yes
Limits	The X an Y limits of the curve in the form '[*xmin, xmax, ymin, ymax*]'	Yes
ShortLabel	A label composed from Name and Suffix but without group name	Yes
XAxis	ID of x-axis attached to curve	Yes
XUnit	Physical type of curve's x-data	Yes
YAxis	ID of y-axis attached to curve	Yes
YUnit	Physical type of curve's y-data	Yes
NumDivisions	Number of divisions in curves data. I.e. the number of separate traces in a group of curves. Groups of curves are usually produced by Monte Carlo analyses	Yes

CurveMarker

An object used to title a curve or mark a feature.

Name	Description	Read Only?
Type	Type of object - always 'CurveMarker'	Yes
Graph	ID of parent graph	Yes
X1	X-data value at arrowhead	No
Y1	Y-data value at arrowhead	No
X2	X position of label in view units relative to arrowhead	No
Y2	Y position of label in view units relative to arrowhead	No
Label	Label of curve marker. May be a symbolic value. Default is %curve:label% which returns the label of the attached curve	No
LabelJustification	Text Alignment. May be one of these values: -1 Automatic 0 Left-Bottom 1 Centre-Bottom 2 Right-Bottom 12 Left-Middle 13 Centre-Middle 14 Right-Middle 8 Left-Top 9 Centre-Top 10 Right-Top	No
Curve	ID of attached curve	No
Division	Division index of attached curve	Yes
Font	Font for label	No
SnapToCurve	'On' or 'Off'. If 'On' marker tracks attached curve i.e its y position is determined by the y value of the curve at the marker's x position. If 'Off' marker may be freely located.	No

FreeText

Free text objects are items of text with no border or background that are not attached to any other object

Name	Description	Read Only?
Type	Type of object - always 'FreeText'	Yes
Graph	ID of parent graph	Yes
X1	X location of object in view units	No
Y1	Y location of object in view units	No
Label	Text displayed. Symbolic values may be used. E.g. %Time% will display the time the object was created.	No
LabelJustification	As CurveMarker (see above) except -1 (automatic) not allowed	No
Font	Font for label	No
Parent	ID of parent object. If text is placed freely on the graph, this will be the same as the Graph property. FreeText objects. however are also used in TextBoxes in which case this returns the id for the TextBox	Yes
Date	Date that the object was created. If the object is on a graph that has been saved to a file then subsequently restored, the date will be the date that the object was originally created.	Yes
Time	Time that the object was created. If the object is on a graph that has been saved to a file then subsequently restored, the time will be the time that the object was originally created.	Yes
Version	Product name and version	Yes

Graph

Name	Description	Read Only?
Type	Type of object - always 'Graph'	Yes
CursorStatusDisplay	Sets method of displaying cursor positions and dimensions. Possible values: Graph Display on graph using CrosshairDimension object StatusBar Display on status bar Both Display on both graph and status bar	No
Path	Path of file to save to. This is the file that will be used by the "File|Save" menu. When saving a graph, this property will be set to the full path name of the file.	No
TabTitle	The text in the title of the tabbed sheet. This can be symbolic. Default is %SourceGroup% %FirstCurve:Label%	No
TitleBar	Text to be displayed in the graph window title bar when the graph's sheet is in view. This may be symbolic. Default is %SourceGroup% (%GroupTitle%)	No
MainCursor	ID of Crosshair object comprising the main cursor. Value = -1 if cursors are not enabled.	Yes

RefCursor	ID of Crosshair object comprising the reference cursor. Value = -1 if cursors are not enabled.	Yes
SourceGroup	The data group that was current when the graph was created	Yes
FirstCurve	ID of the oldest curve on the graph	Yes
GroupTitle	Title of the data group that was current when the graph was created	Yes

The Graph object represents a graph sheet

LegendBox

The LegendBox is used to display labels for every curve on the graph sheet. It consists of a box that is loaded with LegendText objects - one for each curve on the graph. The LegendText objects are automatically loaded when a curve is added to the graph and automatically deleted when a curve is deleted. LegendBox is very similar to the text box and shares the same properties with the following differences and additions:

1. Type property has the value 'LegendBox'
2. LegendBox has one additional property as shown below

Name	Description	Read Only?
LegendLabel	Text of label that is loaded into box when a curve is added to the graph. This can be symbolic in which case it references properties of the LegendText object that displays the text. The default value is %DefaultLabel%	No

LegendText

LegendText objects are used to load legend boxes and cannot be instantiated independently. They are similar to FreeText objects and share the same properties with the following differences and additions:

1. Type property has the value 'LegendText'
2. The Label property is set to the value of the legend box's LegendLabel property when it is added to the box.
3. LegendBox has two additional properties as shown below

Name	Description	Read Only?
Curve	ID of attached curve	Yes
DefaultLabel	The default value for the label. Actually equivalent to %Curve:Label%<n>%Curve:Measurements%. (<n> denotes a new line).	Yes

TextBox

A TextBox consists of a border with a definable background colour into which a FreeText object may be added. TextBox is also the basis of the LegendBox object.

Name	Description	Read Only?
Type	Type of object - always 'TextBox'	Yes
Graph	ID of parent graph	Yes
X1	X location of object in view units	No
Y1	Y location of object in view units	No
X2	Physical width of box in mm. (Ignored if AotoWidth='On')	No
Y2	Physical height of box in mm	No
AutoWidth	'On or 'Off''. If 'On' the width of the box is automatically adjusted according to its contents subject to MaxHeight	No

Colour	Background colour. Either the name of a colour object or a colour specification.	No
Font	Font used for text objects added to the box. In practice this only affects the LegendBox object which is based on TextBox.	No
MaxHeight	Maximum physical height in mm of box. This is only used when AutoWidth='On'	No
Objects	List of object IDs owned by box. Each item is separated by a ';'. A TextBox may only contain one object but the LegendBox object which is based on TextBox may have multiple objects.	Yes

Graph Co-ordinate Systems

Three different units of measure are used to define the location and dimensions of an object on a graph sheet. These are 'View units', 'Physical units' and 'Data units'. These are explained as follows:

'Physical Units' relate to the physical size of the displayed object and have units of millimetres. Physical units are only used for dimensions of some annotation objects and are not used for location. When objects are displayed on a screen an assumption is made for the number of pixels per inch. This depends on the display driver but is typically in the range 75 - 100.

'Data Units' relate to the units of the X and Y axes. Typically an object such as curve marker is located using data units so that it always points to the same point on a curve regardless of how the graph is zoomed or scrolled.

'View Units' relate to the current viewable area of the graph. View units use a co- ordinate system whereby the bottom left of the grid area is co-ordinate (0,0) and the top right corner of the grid is co-ordinate (1,1). View units are used to define the location of objects that need to be at a fixed location on the graph irrespective of zoom magnification.

Event Scripts

There are three special scripts that are automatically called by the simulator system in response to user events. These scripts are detailed in the following table:

on_graph_anno_doubleclick	Called when the user double clicks on certain graph objects
on_accept_file_drop	Called when a file of directory is dropped on a simulator window.

All three scripts are defined internally but can be customised if desired. Details on these event scripts follow.

on_graph_anno_doubleclick

The script is called when some graph objects are double clicked.

The script is passed two arguments when it is called. The first is the object's ID and the second is specific to the object that is double clicked. Currently the second argument is only used by curves and is set to its division index.

on_accept_file_drop

This is called when an a file, folder or group or files and/or folders is dropped on the command shell or graph window.

Two arguments are passed. The first identifies the window type. This may be one of:

Graph	Graph window
Shell	Command shell

The second argument contains a list of full path names of the objects dropped. The items are separated by semi-colons.

User Defined Script Based Functions

Overview

The script language provides a method of creating user defined functions that can be used in any front end expression. These expressions may be used in scripts or on the command line.

User defined functions are used to define some of the goal functions designed for performance and histogram analysis. The scripts for these all begin “uf_” and are registered using the “reg_user_funcs” script. The source for these can be found on the installation CD.

Defining the Function

User defined functions are defined as a script. The arguments to the function and the return value from the function are passed as the script's arguments. The script's first argument is passed by reference and is the return value while the remaining arguments are the arguments passed in the call to the function. The function may have up to seven arguments and they may be of any type. See example below.

Registering the Script

For the expression evaluator to recognise the function name, the script and function name must be registered. This is done with the RegisterUserFunction command. The definition of this is:

RegisterUserFunction *Function-Name Script-Name* [*min-number-args*] [*max-number-args*]

Note that function registration is not *persistent*. That is the registration only lasts for the current session. If you wish to make a permanent function definition, place the RegisterUserFunction command in the startup script.

Example

Here is a trivial example. The following shows the steps to create a function that multiplies a number by 2. First the script

```
Arguments @rv arg1

Let rv = 2*arg1
```

Save this to a file called - say - times_two.sxscr and place it in the script directory.

Now, register the script as a function called “Times2”. To do this, execute the command:

```
RegisterUserFunction Times2 times_two 1 1
```

The definition is now complete. To test it type at the command line:

```
Show Times2(2)
```

You should see the result:

```
Times2(2) = 4
```

Non-interactive and Customised Printing

Overview

The Pulsonix Spice script language provides a number of functions and commands that allow *non-interactive printing*. That is printing without user intervention. This is useful for - say - running multiple simulations in the background and automatically printing the results when the simulation is complete. The same printing facilities may also be used to customise the layout of printed graphs.

The available printing commands are:

ClosePrinter, NewPrinterPage, OpenPrinter. PrintGraph

The functions are:

GenPrintDialog (for interactive printing), **GetPrinterInfo**

Each of these commands and functions is described in detail in its relevant section. Here we give a general overview for the printing procedure.

Procedure

The sequence for a print job is:

1. Open printer. At this stage the printer to be used, page orientation, title and number of copies may be selected.
2. Print pages. The actual graphs to be printed along with scaling and margins are specified here. Any number of pages can be printed.
3. Close printer. This actually starts the physical printing. It is also possible to abort the print job.

Example

Suppose we wish to create a PDF file using "Acrobat Distiller" for the current graph. Of course you can readily do this by selecting **File**|**Print...** and making the appropriate selections using the dialog box. This is no good, however, if you want to create a PDF file for a graph created using an automated simulation, perhaps run overnight. The following script will do this without user intervention.

```
** Get info on system printers
Let printInfo = GetPrinterInfo()

** Search for acrobat distiller. The printer list from GetPrinterInfo
** starts at index 2 so we subtract 2 to get the index
** needed by OpenPrinter
Let distillerIndex = Search(printInfo, 'Acrobat Distiller')-2

** If Acrobat distiller is not on the system
** Search will return -1
if distillerIndex<0 then
    Echo "Acrobat Distiller is not installed"
    exit script
endif

** Open Printer using distiller.
** Orientation will be landscape which is the default
** Number of copies = 1.
** The title will be used by distiller to compose the file name
** for the PDF file i.e. Graph1.PDF
OpenPrinter /title Graph1 /index {distillerIndex}

** Now print the graph
** Major axis on minor axis off. All margins 20mm.
PrintGraph /major on /minor off /margin 20 20 20 20 /caption "Test Print"

** Close Printer. This will actually start the print
ClosePrinter
```

You can of course replace 'Acrobat Distiller' with any printer that is on your system. You must use the printer's name as listed in the Printers section of the system control panel. You can also find a list of system printers from within Pulsonix Spice by typing at the command line:

```
Show GetPrinterInfo()
```

The first two values are numbers but the remaining are the currently installed printers on your system.

If you omit /index switch for the `OpenPrinter` command, the application default printer (not the *system* default printer) will be used. The application default printer is the same as the system default printer when Pulsonix Spice starts but will change whenever you select a different printer using the **File** menu and **Print...** option.

Creating and Modifying Toolbars

From version 2, Pulsonix Spice allows the complete customisation of toolbars. You can modify the definitions of existing toolbars and buttons, as well as create new toolbars and new tool buttons. This section explains how.

Modifying Existing Toolbars and Buttons

You can rearrange the button layout of existing toolbars by modifying the 'Set' option variables that define them.

The following table shows the name of the 'Set' variable to use for each one.

'Set' Variable Name	Toolbar
CommandShellMainButtons	CommandShellMain
CommandShellMainNoSchemButtons	CommandShellMain
GraphMainButtons	GraphMain

The 'Set' variable should be set to a value consisting of a semi-colon delimited list of valid button names.

To determine the current definition, use the GetOption function with the 'Set' variable name as described in the table above.

Redefining Button Commands

You can change the command executed when a button is pressed using the command DefButton.

You can redefine any of the pre-defined buttons.

Defining New Buttons and Editing Buttons

You can define completely new buttons with your own graphic design and add them to an existing toolbar. The same method can also be used to redefine the graphics for existing buttons.

This is done using the command CreateToolButton. These are the steps to take:

1. Create a graphical image for the button. This should be in a windows bitmap (.bmp), portable network graphic (.png) or JPEG (.jpg) format. You can use almost any paint application to do this. But, if you want to define a mask - that is you wish to define transparent areas - then you must use an editor capable of creating 'portable network graphics' (PNG) images. However, this is rarely necessary in practice and none of the built in graphics define a mask. This is because the simulator will automatically create one that makes the area outside the perimeter of the image transparent. The result is usually satisfactory.
   You can make your graphic any size, but to be compatible with the built-in images, you should make them 16x16 pixels. The built-in graphics are all 16 colour, but you can use any colour depth supported by your system.
   When you have created your image, you should save or copy it to the images directory. This is located at:

   `…\support\images`

2. Execute the command CreateToolButton. As with menu and key definitions, the definitions created by this command are not persistent that is they will be lost when Pulsonix Spice exits. To make permanent definitions, you should place the commands in the start up script. See Startup Script for more details.
   CreateToolButton will not add the button to any toolbar nor does it assign a command to be executed when it is pressed. These operations are described in the following steps.

3. Define a command to be executed when this button is pressed. This is done using the command DefButton. Again, this should be place in your startup script.
4. Add the button to a toolbar. See “Modifying Existing Toolbars and Buttons” to find out how to add this to an existing toolbar. If you wish to create a new toolbar for the new button, see “Creating New Toolbars” below.

Creating New Toolbars

To create a completely new toolbar, use the command Create ToolBar. This will create an empty toolbar.

To add buttons to a new toolbar, you must use the command Define ToolBar. You can add both pre-defined and user-defined buttons to a custom toolbar.

Pre-defined Buttons

The following table lists all the buttons that are pre-defined. All of these buttons may be redefined if required.

The bitmaps are embedded in the program binary, but can also be found on the install CD in the directory script/images.

The command executed by each button can be found using the command ListStdButtonDefs.

Index

B

Braced substitutions, 26

C

Collections, 31
Colours
  ChooseColour command, 135
Commands
  About, 132
  AddLegendProp, 134
  Cd, 135
  ChooseColour, 135
  Close, 135
  CloseGraphSheet, 135
  CursorMode, 137
  DefItem, 140
  DefKey, 144
  DefMenu, 146
  DelCrv, 148
  DeleteAxis, 148
  DelLegendProp, 149
  DelMenu, 149
  Display, 149
  Execute, 150
  Focus, 151
  Font, 151
  GraphZoomMode, 151
  Help, 151
  HideCurve, 152
  KeepGroup, 152
  Let, 152
  ListModels, 154
  ListStdKeys, 154
  ListStdMenu, 154
  LoadModelIndex, 154
  MakeAlias, 155
  MakeCatalog, 155
  Mcd, 155
  Md, 156
  MessageBox, 156
  NewAxis, 156
  NewGraphWindow, 156
  NewGrid, 156
  NoPaint, 156
  Pause, 158
  PlaceCursor, 158
  Plot, 158
  Quit, 160
  Rd, 160
  Redirect, 162
  RegisterDevice, 162
  Reset, 163
  RestDesk, 163
  Resume, 163
  SaveDesk, 165
  SaveRhs, 166
  ScriptAbort, 166
  ScriptPause, 167
  ScriptResume, 167
  ScriptStep, 167
  SelectCursorMode, 167
  SelectCurve, 168
  SetCurveName, 168
  SetGroup, 169
  SetRef, 169
  SetUnits, 169
  ShowCurve, 171
  SizeGraph, 171
  Title, 172
  Trace, 172
  UndoGraphZoom, 172
  Unlet, 173
  Wait, 173

D

Defining keys, 175
  DefKey command, 144
  ListStdKeys command, 154
Defining menus, 175
  DefMenu command, 146
  DelMenu command, 149
  ListStdMenu command, 154

E

Exporting data, 183
Expressions
  command line, 26
  in scripts, 24

F

Functions
  AddRemoveDialog, 33
  BoolSelect, 33
  ChooseDir, 33
  EditSelect, 33
  GetFilecd, 33
  InputGraph, 33
  RadioSelect, 33
  ValueDialog, 33

G

Global variables, 23
Graphs
  plotting
  Plot command, 158
Groups, 30
  current, 30

I

Importing data, 183

M

Menus
  Simulator|Change Data Group..., 30
  Simulator|Keep Current Data Group, 30

N

New line character, 22

P

Physical type, 33
Properties
  graph legend
  AddLegendProp command, 134

R

Reference, vector, 33

S

Safe Operating Area Testing, 82
Scripts, 15
  braced substitutions, 26
  bracketed lists, 26
  built-in, 36
  commands, 27
  constants, 21
  controlling execution, 34
  ScriptPause command, 167
  ScriptResume command, 167
  ScriptStep command, 167
  debugging, 36
  empty data, 23
  errors, 34
  executing, 35
  exit statements, 30
  expressions, 24
  for statement, 29
  functions, 25
  global variables, 23
  if statement, 28
  loops, 15, 16, 17
  new line, 22
  operator precedence, 25
  tutorial, 15
  type conversion, 27
  types, 21
  variables, 21
  vectors, 22
  while statement, 29
SOA, 82

T

Type conversion (scripts), 27

V

Vector reference, 33
vectors, 22