The Signal script language

Version 3

Version 3.00	Sep 2004
Version 3.01	Dec 2004
Version 3.02	Feb 2005
Version 3.03	Apr 2005
Version 3.04	Sep 2005
Version 3.06	Jan 2006
Version 3.07	Apr 2006
Version 3.09	Feb 2007
Version 3.10	Sep 2007

Published by:

Cambridge Electronic Design Limited
Science Park
Milton Road
Cambridge
CB4 0FE
UK

Telephone:	Cambridge (01223) 420186
	International +44 1223 420186
Fax:	Cambridge (01223) 420488
	International +44 1223 420488
Email:	info@ced.co.uk
Web site:	http://www.ced.co.uk

Curve fitting procedures are based on routines in Numerical Recipes: The Art of Scientific Computing, published by Cambridge University Press, and are used by permission.

Table of Contents

1 Introduction

What is a script?

For many users, the interactive nature of Signal may provide all the functionality required. This is often the case where the basic problem to be addressed is to present the data in a variety of formats for visual inspection with, perhaps, a few numbers to extract by cursor analysis and a picture to cut and paste into another application. However, some users need analysis of the form:

1. Find the first peak in the frame after a cursor.
2. Find the trough after that.
3. Compute the time difference between these two points and the slope of the line.
4. Print the results.
5. If not at the end of the file, move to the next frame and go back to step 1.

This could all be done manually, but it would be very tedious. A script can automate this process, however it requires more effort initially to write it. A script is a list of instructions (which can include loops, branches and calls to user-defined and built-in functions) that control the Signal environment. You can create scripts by example, or type them in by hand. If you are new to script writing it would be a good idea to work through the relevant chapters in the Training Manual before referring to this manual for more detailed information.

Views and view handles

The most basic concept in a script is that of a view and the view handle that identifies it. A view is a window in Signal that the script language can manipulate

There is always a *current view*. Even if you close all windows the Log view, used for text output by the `PrintLog()` command, remains. Whenever you use a built-in function that creates a new view, the function returns a *view handle*. The view handle is simply an integer number that identifies the view. It is used with the `View()` and `FrontView()` functions to specify the current view and the view that should be on top of all windows.

The running script is hidden from most commands, however you can obtain its handle using `App()` so you can show and hide it.

Whenever a script creates a new view, the view becomes the current view. However, views are created invisibly so that they can be configured before appearing. You can use `WindowVisible(1)` to display a new window.

Writing scripts by example

To help you write scripts Signal can monitor your actions and write the equivalent script. This is often a great way to get going writing scripts, but it has limitations. Scripts generated this way only repeat actions that you have already made. The good point of recording your actions is that Signal shows you the correct function to use for each activity.

For example, let us suppose that you have opened a data file. Use the Turn Recording On option of the Script menu. Click on the data file view, then select Analysis, New memory view, Waveform average, and with the default settings, process all the frames in the file. Finally you use the Stop recording command in the Script menu. Signal opens a new window holding the equivalent script:

```
var v3%;
var v4%;
v3%:=ViewFind("Example.cfs");
FrontView(v3%);
v4%:=SetAverage(-1,0.04,0,0,0);
WindowVisible(1);
ProcessFrames(1,-1,-1,0,1);
```

The `ViewFind()`, `FrontView()`, `SetAverage()`, and `ProcessFrames()` functions are described in this manual and they reflect the actions that you performed. The `v3%` and `v4%` variables hold view handles. The script needs to save these handles in unique variables. To do this it generates variable names based on the internal view number.

The `WindowVisible(1)` command is present because new windows are hidden when they are created by the script. Signal creates invisible windows so that you can size and position them before display to prevent excessive screen repainting.

The script recorder produces all the optional arguments for `ProcessFrames()`, to process all the frames from 1 to the last frame in the file, and to optimise the y axes after processing. The memory view is not cleared before processing, which in this case makes no difference as the new memory view is created with zero data.

Now do the same again, using the Turn Recording On option of the Script menu, clicking on the data file view, selecting Analysis, New memory view, Waveform average, as before, but this time change the settings to select channel 3, width 0.02 and start offset of 0.01, then process. When you use the Stop recording command you will see a similar script, but with different arguments for `SetAverage()`. In this example we did not change the options in the Process dialog.

```
var v5%;
var v8%;
v5%:=ViewFind("Example.cfs");
FrontView(v8%);
v4%:=SetAverage(-1,0.02,0.01,0,0);
WindowVisible(1);
ProcessFrames(1,-1,-1,0,1);
```

You can use the Turn Recording On option of the Script menu before any small sequence of operations. Then use the Stop recording command in the Script menu to see the script commands generated.

Using recorded actions

You can now run the recorded script, using the control buttons at the upper right of the script window. The script runs and generates a new memory view, repeating your actions. Now suppose we want to run this for several files, each one selected by the user. You must edit the script a bit more and add in some looping control. The following script suggests a solution. Notice that we have now changed the view handle variables to names that are a little easier to remember.

We simplify the `ProcessFrames()` command to replace start frame by -1 for all frames in the data file. Without the optional arguments, the y axis will not be optimised after the processing. `SetAverage(1)` also needs no extra arguments to average data in channel 1 for the whole frame.

```
var fileH%, aveH%;               'view handle variables
fileH% := FileOpen("", 0, 1);    'blank for dialog, single window
while fileH% > 0 do              'FileOpen returns -ve if no file
   aveH% := SetAverage(1);
   WindowVisible(1);                    'Average channel 1
   ProcessFrames(-1);            'Process all frames in the file
   Draw();                       'Update the average display
   fileH% := FileOpen("", 0, 1); 'ask for the next file, or cancel
wend;
```

This time, Signal prompts you for the file to open. The file identifier is negative if anything goes wrong opening the file, or if you press the Cancel button. We have also included a `Draw()` statement to force Signal to draw the data after it calculates the average. There is a problem with this script if you open a file that does not contain a channel 1 that holds waveform data although this is unlikely in Signal. We will deal with this a little later.

However, you will find that the screen gets rather cluttered up with windows. We do not want the original window once we have calculated the average, so the next step is to delete it, adding the line

```
View(fileH%).FileClose();         'Shut the old window
```

The `View().` syntax allows a function to access data belonging to a view other than the current view. The `fileH%` argument, and the dot after the command, tell the script system that we want to change the current view to the data file view temporarily, for the duration of the `FileClose()` function.

We have also added a line to close down all the windows at the start, to reduce the clutter when the script starts.

```
var fileH%, aveH%;
FileClose(-1);                   'close all windows to tidy up
fileH% := FileOpen("", 0, 1);    'use a blank name to open dialog
while fileH% > 0 do              'FileOpen returns -ve if no file
  aveH% := SetAverage(1);         'set up average on selected chan
  WindowVisible(1);               'make average visible
  ProcessFrames(-1);              'do the average
  View(fileH%).FileClose();       'Shut the old window
  Draw();                         'Update the average display
  fileH% := FileOpen("", 0, 1);   'ask for the next file, or cancel
wend;
```

This seems somewhat better, but we still have the problem that there will be an error if the file does not hold a channel 1, or it is of the wrong type. The solution to this is to ask the user to choose a channel using a dialog. We will have a dialog with a single field that asks us to select a suitable channel:

```
var fileH%, aveH%, chan%;         'Add a new variable for channel
FileClose(-1);                    'close all windows to tidy up
fileH% := FileOpen("", 0, 1);     'use a blank name to open dialog
while fileH% > 0 do               'FileOpen returns -ve if no file
  DlgCreate("Channel selection"); 'Start a dialog
  DlgChan(1, "Choose channel to average", 1); 'all waveform
  if (DlgShow(chan%) > 0) and     'User pressed OK and...
    (chan% > 0) then              '...selected a channel?
    aveH% := SetAverage(chan%);   'set up average on selected chan
    WindowVisible(1);             'make average visible
    ProcessFrames(-1);            'average all the frames
    View(fileH%).FileClose();     'Shut the old window
    Draw();                       'Update the display
  endif
  fileH% := FileOpen("", 0, 1);   'ask for the next file
wend;
```

The `DlgCreate()` function has started the definition of a dialog with one field that the user can control. The `DlgChan()` function sets a prompt for the field, and declares it to be a channel list from which we must select a channel (or we can select the No channel entry). The `DlgShow()` function opens the dialog and waits for you to select a channel and press OK or Cancel. The `if` statement checks that all is well before making the histogram.

Derived views

The current view when the `ProcessFrames()` command is used is the memory view and we may want to access information about the data file, such as the maximum frame number in the original time view. The `View().` syntax allows a function to access data belonging to a view other than the current view.

```
var fileV%;
var aveV%;
fileV%:=ViewFind("Example.cfs");
FrontView(fileV%);
aveV%:=SetAverage(-1,0.04,0,0,0);
WindowVisible(1);
ProcessFrames(1, View(fileV%).FrameCount(),-1,0,1);
```

In this example we replaced -1 for last frame in file with the actual frame number returned by the `FrameCount()` function. The `fileV%` argument, and the dot after the command, tell the script system that we want to change the current view to the data file view temporarily, for the duration of the `FrameCount()` function.

In many scripts we will have a variable such as `fileV%` holding the data view handle, but you can also use the function `ViewSource()` to access it directly. The following script shows how you would ensure that when you present this message you are counting frames in the data view associated with the current memory view.

```
var fileV%, aveV%;
fileV%:=ViewFind("Example.cfs");  'view data file
if fileV%>0 then
 FrontView(fileV%);
 aveV%:=SetAverage(3);            'set up average of channel 3
 WindowVisible(1);
 ProcessFrames(-1);        'process all frames in the file
 Message("We averaged %d frames",View(ViewSource()).FrameCount());
endif
```

In this example the `Message()` command displays a string in which `%d` is replaced by the value for the frame count.

Notation conventions

Throughout this manual we use the font of this sentence to signify descriptive text. Function declarations, variables and code examples print in a monospaced font, for example `a% := View(0)`. We show optional keywords and arguments to functions in curly braces:

```
func Example(needed1, needed2 {,opt1 {,opt2}});
```

In this example, the first two arguments are always required; the last two are optional. Any of the following would be acceptable uses of the function:

```
a := Example(1,2);         'Call omitting the optional arguments
a := Example(1,2,3);       'Call omitting one argument
a := Example(1,2,3,4);     'Call using all arguments
```

A vertical bar between arguments means that there is a choice of argument type:

```
func Choice( i%|r|str$ );
```

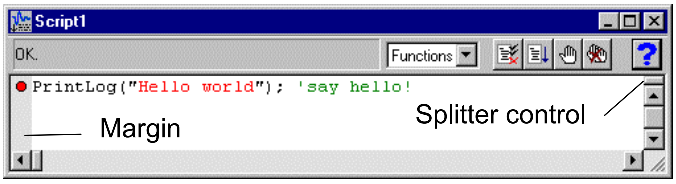
In this case, the function takes a single argument that could be an integer, a real or a string. The function will detect the type that you have passed and may perform a different action depending upon the type.

Three dots (...) stand for a list of further, similar items.

2 The script window and debugging

Script window

You use the script window when you write and debug a script. Once you are satisfied that your script runs correctly you would normally run a script from the script menu without displaying the source code. You can have several scripts loaded at a time and select one to run with the Script menu Run Script command.



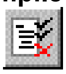
The script window is a text window with a few extra controls including a “splitter” control so that you can view two parts of the script at the same time. To use the splitter control, drag it down the window. To cancel it, drag it to the top or bottom of the window. There is no script language control of the splitter.

To the left of the text area is a margin where you can set break points (one is shown already set) and bookmarks, and where the current line of the script is indicated during debugging. Above the text area is a message bar and several controls. The controls have the following functions:

Functions

This control is a quick way to find any `func` or `proc` in your script. Click on this to display a list, in alphabetical order, of the names of all user-defined routines. Select one, and the window will scroll to it. To be located, the keywords `func` and `proc` must be at the start of a line and the name of the routine must be on the same line.

Compile

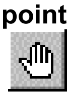


The script compiler checks the syntax of the script and if no errors are found it creates the compiled version, ready to run. If the script has not been changed since the last compile and no other script has been compiled, the button is disabled, as there is no need to compile again. Signal can have one compiled script in memory at a time.

Run

If the script has not been compiled it is compiled first. If no errors are found, Signal runs the compiled version, starting from the beginning. Signal skips over `proc ... end;` and `func ... end;` statements, so the initial code can come before, between or after any user-defined procedures and functions. This button is disabled once the script has started to run.

Set break point



This button sets a break point on the line containing the text caret, or clears a break if one is already set. A break point stops a running script when it reaches the start of the line containing the break point. You can also set and clear break points by moving the mouse pointer over the margin on the left of the script and double clicking.

Not all statements can have break points set on them. Some statements, such as `var`, `const`, `func` and `proc` compile to entries in a symbol table; they generate no code. If you set a break point on one of them the break point will appear at the next statement that is breakable. If you set break points before you compile your script, you may find that some break points move to the next “breakable” line when you compile.

Clear all break points

This button is enabled if there are any break points set in the script. Click this button to remove all break points from the script. Break points can be set and cleared at any time, even before your script has been compiled.

Help

This button provides help on the script language. It takes you to an alphabetic list of all the built-in script functions. If you scroll to the bottom of this list you can also find links to the script language syntax and to the script language commands grouped by function. Within a script, you can get help on keywords and built in commands by clicking on the keyword or command and pressing the `F1` key.

Syntax colouring

Signal version 3 supports syntax colouring for the script editor. The language keywords have the standard colour blue, quoted text strings have the standard colour red, and comments have the standard colour green. Normal text is black and the text background is white.

Editing features for scripts

There are some extra editing features that can help you when writing scripts. If you like to indent your text, you can indent and outdent selected blocks by selecting one or more characters, then use the `Tab` key to indent and the `Shift-Tab` combination to outdent. The text moves by one tab stop.

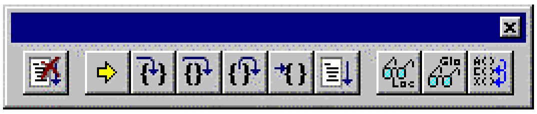
Despite all our attempts to make writing a script easy, and all your attempts to get things right, sooner or later (usually sooner), a script will refuse to do what you expect. Rather than admit to human error, programmers attribute such failures to "bugs", hence the act of removing such effects is "debugging". The term dates back to times when an insect in the high voltage supply to a thermionic valve really could cause hardware problems.

To make bug extermination a relatively simple task, Signal has a "debugger" built into the script system. With the debugger you can:

- Step one statement at a time
- Step into or over procedures and functions
- Step out of a procedure or function
- Step to a particular line
- View local and global variables
- Edit variable values
- See how you reached a particular function or procedure
- Set and clear break points

With these tools at your disposal, most bugs are easy to track down.

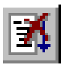
Preparing to debug

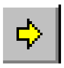


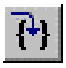
Unlike most languages, the Signal script language does not need any special preparation for debugging except that you must set a break point or include the `Debug();` command at the point in your script at which you want to enter the debugger. When your script reaches the break point or the `Debug();` command, the script stops and the debug toolbar will appear (if it was not already visible). The picture above shows the toolbar as a floating window, but you can dock it to any side of the Signal window by dragging it over to the window edge.

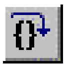
You can also enter the debugger by pressing the `Esc` key (you may need to hold it down for a second or two, depending on what the script is doing). If the `Toolbar()` or `Interact()` commands are active, hold down the `Esc` key and click on a button. This is a very useful way to break out of programs that are running round in a loop with no exit! You can stop the user entering debug with the `Debug(0)` command, but we suggest that this feature is added after the script has been tested! Once you have disabled debugging, there is no way out of a loop.

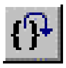
The final way to enter the debugger is when the preferences dialog (accessed from the Edit menu) is used to set scripts to enter the debugger when a runtime error occurs. A runtime error may be something like a division by zero.

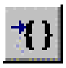

Stop running the script. There is no check that you really meant to do this, as we assume that if you know enough to get into the debugger, you know what you are doing! You can use the `Debug()` command to disable the debugger.

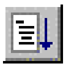

Display the current line in the script. If the script window is not visible, this will make it visible, then bring it to the top and scroll the text to the current line.

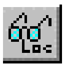

If the current statement contains a call to a user-defined `Proc` or a `Func`, step into it, otherwise just step. This does not work with the `Toolbar()` command which is not user-defined, but which can cause user-defined routines to be called. To step into a user-defined `Func` that is linked to a `Toolbar()` command, set a break point in the `Func`.


Step over this statement to the next statement. If you have more than one statement on a line you will have to click this button once for each statement, not once per line.

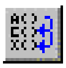

If you are in a procedure or function, step until you return from it. This does not work if you are in a function run from the `Toolbar()` command as there is nowhere to return to. In this case, the button behaves as if you had pressed the run button.


Run until the script reaches the start of the line containing the text caret. This is slightly quicker than setting a break point, running to it, then clearing it (which is what this does).

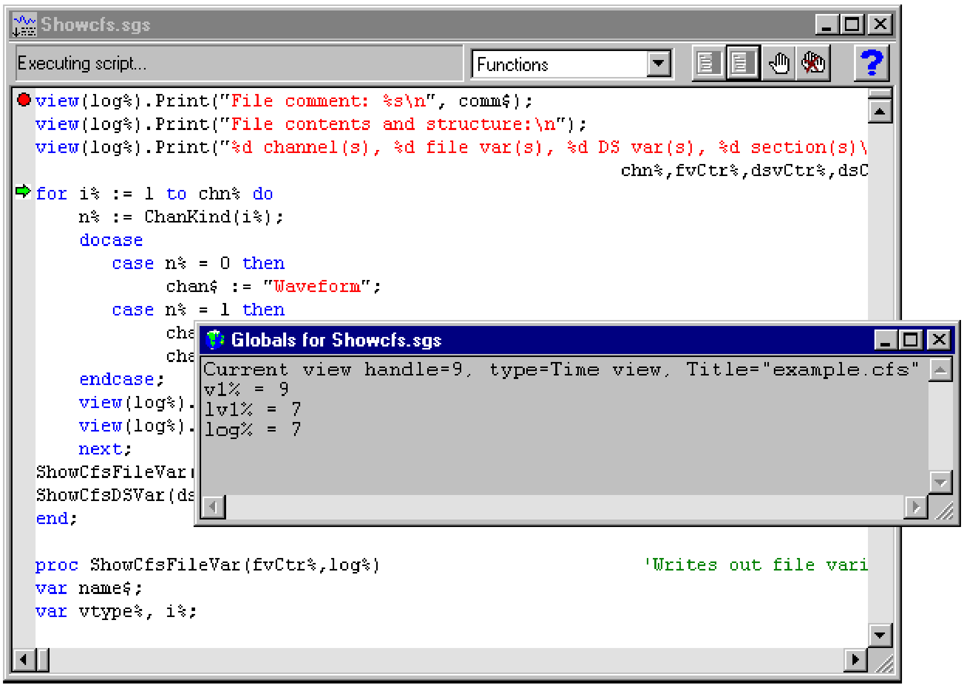

Run the script. This disables the buttons on the debug toolbar and the script runs until it reaches a break point or the end of the script.


Show the local variables for the current user-defined `func` or `proc` in a window. If there is no current routine, the window is empty. You can edit a value by double clicking on the variable. Elements of arrays are displayed for the width of the text window. If an array is longer than the space in the window, the text display for the array ends with … to show that there is more data.


Show the global variable values in a window. You can edit a global variable by double clicking on it. The very first entry in this window lists the current view by handle, type and window name.


Display the call stack (list of calls to user-defined functions with their arguments) on the way to the current line in a window. If the `Toolbar()` function has been used, the arguments for it appear, but the function name is blank.

The debug toolbar and the locals, globals and the call window close at the end of a script. The buttons in the debug toolbar are disabled if they cannot be used. If you forget what a particular button does, move the mouse pointer over the button. A “Tool tip” window will open next to the button with a short description and if the Status bar is visible, a longer description can be seen there.

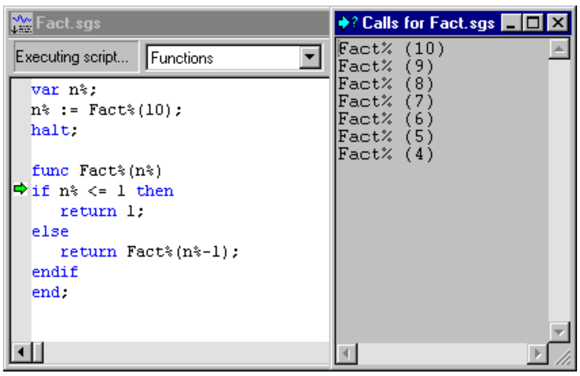


Inspecting variables

If the locals or globals windows are open, they display a list of variables. If there are more variables than can fit in the window you can scroll the list up and down to show them all. Simple variables are followed by their values. If you double click on one a new window opens in which you can edit the value of the variable.

If you double click on an array, a new window opens that lists the values of the elements of the array. You must double click on an element to edit the value. There is a limit of 32000 to the number of array elements that can be displayed and edited. This should not be a problem for most users. Function and procedure arguments show the name and cannot be edited.

Call stack



The call stack can sometimes be useful to figure out how your script arrived at a position in your code. This is particularly true if your script makes recursive use of functions. A function is recursive when it calls itself, either directly, or indirectly through other functions. A common fault with scripts is to have mutually recursive user options. This leads to users burrowing deeper and deeper into the call stack until they run out of memory. The call stack can help to detect such problems.

3 Script language syntax

Script format

A script consists of lines of text. Each line can be up to 240 characters long, however we suggest a maximum line length of 78 characters as experience shows that this makes printing and transfer of scripts to other systems simple.

The script compiler treats consecutive white space as a single space except within a literal string. White space characters are end of line, carriage return, space and tab. The compiler treats comments as white space.

The maximum size of a script is limited by the number of instructions that it compiles into. This number is displayed in the status bar of the script window when you compile. The limit is currently 1,000,000 instructions, which is a very large script, probably around 160,000 lines of typical script code.

Keywords and names

All keywords, user-defined functions and variable names in the script language start with one of the letters `a` to `z` followed by the characters `a` to `z` and `0` to `9`. Keywords and names are not case sensitive, however users are encouraged to be consistent in their use of case as it makes scripts easier to read. Variables and user-defined functions use the characters `%` and `$` at the end of the name to indicate integer and string type.

User-defined names can extend up to a line in length. Most users will restrict themselves to a maximum of 20 or so characters.

The following keywords are reserved and cannot be used for variables or function names:

```
and       band      bor       bxor      case
const     diag      do        docase    else
end       endcase   endif     for       func
halt      if        mod       next      not
or        proc      repeat    resize    return
step      then      to        trans     until
var       view      wend      while     xor
```

Further, names used by Signal built-in functions cannot be redefined as user functions or global variables. They can be redefined as local variables (not recommended).

Data types

There are three basic data types in the script language: real, integer and string. The real and integer types store numbers; the string type stores characters. Integer numbers have no fractional part, and are useful for indexing arrays or for describing objects for which fractions have no meaning. Integers have a limited (but large) range of allowed values.

Real numbers span a very large range of number and can have fractional parts. They are often used to describe real-world quantities, for example the weight of an object.

Strings hold text and automatically grow and shrink in length to suit the number of text characters stored within them.

Real data type

This type is a double precision floating point number. Numbers are stored to an accuracy of at least 16 decimal digits and can have a magnitude in the range 10^{-308} to 10^{308}. Variables of this type have no special character to identify them. Real constants have a decimal point or the letter `e` or `E` to differentiate from integers. White space is not allowed in a sequence of characters that define a real number. Real number constants have one of the following formats where `digit` is a decimal digit in the range 0 to 9:

```
{-}{digit(s)}digit.{digit(s)}{e|E{+|-}digit(s)}
{-}{digit(s)}.digit{digit(s)}{e|E{+|-}digit(s)}
{-}{digit(s)}digitE|e{+|-}digit(s)
```

A number must fit on a line, but apart from this, there is no limit on the number of digits. The following are legal real numbers:

```
1.2345 -3.14159 .1 1. 1e6 23e-6 -43e+03
```

`E` or `e` followed by a power of 10 introduces exponential format. The last three numbers above are: `1000000 0.000023 -43000.0`. The following are not real constants:

`1 e6`	White space is not allowed	`1E3.5`	Fractional powers are not allowed
`2.0E`	Missing exponent digits	`1e500`	The number is too large

Integer data type

The integer type is identified by a `%` at the end of the variable name and stores 32-bit signed integer (whole) numbers in the range -2,147,483,648 to 2,147,483,647. There is no decimal point in an integer number. An integer number has the following formats (where `digit` is a decimal digit 0 to 9, and `hexadecimal-digit` is 0 to 9 or `a` to `f` or `A` to `F`, with `a` standing for decimal 10 to `f` standing for decimal 15):

```
{-}{digit(s)}digit
{-}0x|X{hexadecimal-digit(s)}hexadecimal-digit
```

You may assign real numbers to an integer, but it is an error to assign numbers beyond the integer range. Non-integral real numbers are truncated (towards zero) to the next integral number before assignment. Integer numbers are written as a list of decimal digits with no intervening spaces or decimal points. They can optionally be preceded by a minus sign. The following are examples of integers:

```
1 -1 -2147483647 0 0x6789abcd 0X100 -0xced
```

Integers use less storage space than real numbers and are slightly faster to work with. If you do not need fractional numbers or huge numeric ranges, use integers.

String data type

Strings are lists of characters. String variable names end in a `$`. String variables can hold strings up to 65534 characters long. Literal strings in the body of a program are enclosed in double quotation marks, for example:

```
"This is a string"
```

A string literal may not extend over more than one line. Consecutive strings with only white space between them are concatenated, so the following:

```
"This string starts on one lin"
"e and ends on another"
```

is interpreted as `"This string starts on one line and ends on another"`. Strings can hold special characters, introduced by the escape character backslash:

`\"`	The double quote character (this would normally terminate the string)
`\\`	The Backslash character itself (beware DOS paths)
`\t`	The Tab character
`\n`	The New Line character (or characters, depending on the system)
`\r`	The Carriage Return character (ASCII code 13)

Conversion between data types

You can assign integer numbers to real variables and real numbers to integer variables (unless the real number is out of the integer range when a run-time error will occur). When a real number is converted to an integer, it is truncated. The `Asc()`, `Chr$()`, `Str$()` and `Val()` functions convert between strings and numbers.

Arrays of data

The three basic types (integers, reals and strings) can be made into arrays with from 1 to 5 dimensions. Before Signal version 3.07, the maximum number of dimensions allowed was 2). We call a one-dimensional array a vector and a two-dimensional array a matrix to match common usage. Declare arrays with the `var` statement:

```
var v[20], M[10][1000], nd[2][3][4][5][6];
```

This declares a vector with 10 elements, a matrix with 10 rows and 1000 columns and a 5-dimensional array with 720 elements. To reference array elements, enclose the element number in square brackets (the first element in each dimension is number 0):

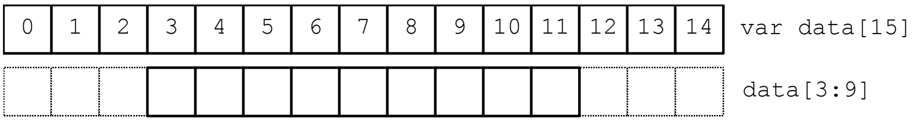
```
v[6] := 1; x := M[8][997]; nd[1][0][0][0][2] := 4.5;
```

Vector subsets

Use `v[start:size]` to pass a vector or a subset of a vector `v` to a function. `start` is the index of the first element to pass, `size` is the number of elements. Both `start` and `size` are optional. If you omit `start`, 0 is used. If you omit `size`, the sub-set extends to the end of the vector. To pass the entire vector use `v[0:]`, `v[:]`, `v[]` or just `v`.

For example, consider the vector of real numbers declared as `var data[15]`. This has 15 elements numbered 0 to 14. To pass it to a function as a vector, you could specify:

`data` or `data[]`	This is the entire vector. This is the same as `data[:]` or `data[0:15]`.
`data[3:9]`	This is a vector of length 9, being elements 3 to 11.
`data[:8]`	This is a vector of length 8, being elements 0 to 7.

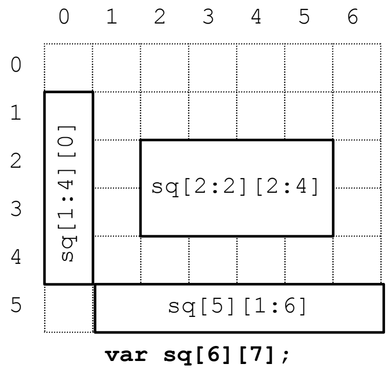


Matrix subsets

With a matrix you have more options. You can pass a single element, a vector sub-set, or a matrix sub-set. Consider the matrix of real numbers defined as `var sq[6][7]`. You can pass this as a vector or a matrix to a function as (`a`, `b`, `c` and `d` are integer numbers):

`sq[a][b:c]`	a vector of length c
`sq[a][]`	a vector of length 7
`sq[a:b][c]`	a vector of length b
`sq[][c]`	a vector of length 6
`sq[a:b][c:d]`	a matrix of size `[b][d]`
`sq` or `sq[][]`	a matrix of size `[6][7]`

This diagram shows how sub-sets are constructed. `sq[1:4][0]` is a 4 element vector. This could be passed to a function that expects a vector as an argument. `sq[5][1:6]` is a vector with 6 elements. `sq[2:2][2:4]` is a matrix, of size `[2][4]`.



N-Dimensional array subsets

With more than 2 dimensions, you can make a subset of any number of dimensions up to the size of the original array. These examples show some of the possibilities for passing a source array with 5 dimensions defined as `var nd[4][5][6][7][8]`;

`nd` or `nd[][][][][]`	The entire 5 dimensional array
`nd[0][][][][]`	A 4 dimensional array of size `[5][6][7][8]`
`nd[1:2][][2][][]`	A 4 dimensional array of size `[2][5][7][8]`
`nd[1][2][3][4][]`	A vector of size `[8]`
`nd[][][0][0][0]`	A matrix of size `[4][5]`

Transpose of a vector or matrix

You can pass the transpose of a vector or matrix to a function with the `trans()` operator, or by adding ` (back-quote) after the array or matrix name. The transpose of a matrix swaps the rows and columns. To be consistent with normal matrix mathematics, a one-dimensional array is treated as a column vector and is transposed into a matrix with 1 row. That is given `var data[15]`, `trans(data)` is a matrix of size `[1][15]`.

```
var M[5][3], v[5], W[5][5];
PrintLog(M, M`);              'Print M and its transpose
PrintLog(M[][], trans(M[][])); 'Exactly the same as last line
MatMul(W, M, M[][]`);         'set W to M times its transpose
MatMul(W, v, v`);             'set W to v times its transpose
```

From Signal version 3.07 onwards you can apply the transpose operator to arrays of higher dimensions. The result is an array with the dimensions and indexing reversed. That is, the transpose of `x[2][3][4]` is an array of size `[4][3][2]`. The element `x[i][j][k]` in the original becomes the element at index `[k][j][i]` in the transposed array.

Diagonal of a matrix

You can pass the diagonal of a matrix to a function using the `diag()` operator. This expects a matrix as an argument and produces a vector whose length is the smaller of the dimensions of the matrix. Given a matrix `M[10][10]`, `diag(M)` is a 10 element vector.

From version 3.07 of Signal you can take the diagonal of any array with more than 1 dimension. The result is a vector with the length of the smallest dimension of the array. For example, given `var a[4][5][6]`, `diag(a)` is a vector of length 4 holding the elements: `a[0][0][0]`, `a[1][1][1]`, `a[2][2][2]` and `a[3][3][3]`.

Data views as arrays

The script language treats a data view as vectors of real numbers, one vector per channel. To access a vector element use `View(v%,ch%).[index]` where `v%` is the view, `ch%` is the channel and `index` is the bin number, starting from 0. You can pass a channel as an array to a function using `View(v%, ch%).[]`, or `View(v%, ch%).[a:b]` to pass a vector subset starting at element `a` of length `b`. You can omit `ch%`, in which case channel 1 is used. You can also omit `View(v%,ch%)`, in which case channel 1 in the current view is used. See the `View()` command for more information.

If you change a visible data view, the modified area is marked as invalid and will update at the next opportunity.

Statement types

The script language is composed of statements. Statements are separated by a semicolon. Semicolons are not required before `else`, `endif`, `case`, `endcase`, `until`, `next`, `end` and `wend`, or after `end`, `endif`, `endcase`, `next` and `wend`. White space is allowed between items in statements, and statements can be spread over several lines. Statements may include comments. Statements are of one of the following types:

- A variable or constant declaration
- An assignment statement of the form:

`variable := expression;`	Set the variable to the value of the expression
`variable += expression;`	Add the expression value to the variable
`variable -= expression;`	Subtract the expression value from the variable
`variable *= expression;`	Multiply the variable by the expression value
`variable /= expression;`	Divide the variable by the expression value

  The `+=`, `-=`, `*=` and `/=` assignments were added at version 3.02. `+=` can also be used with strings (`a$+=b$` is the same as `a$:=a$+b$`, but is more efficient).
- A flow of control statement, described below
- A procedure call or a function with the result ignored, for example `View(vh%);`

Comments in a script

A comment is introduced by a single quotation mark. All text after the quotation mark is ignored up to the end of the line.

```
View(vh%);  'This is a comment, and extends to the end of the line
```

Variable declarations

Variables are created by the `var` keyword. This is followed by a list of variable names. You must declare all variable names before you can use them. Arrays are declared as variables with the size of each dimension in square brackets. The first item in an array is at index 0. If an array is declared as being of size `n`, the last element is indexed `n-1`.

```
var myInt%,myReal,myString$;      'an integer, a real and a string
var aInt%[20],arl[100],aStr$[3]   'integer, real and string vectors
var a2d[10][4];                   '10 rows of 4 columns of reals
var square$[3][3];                '3 rows of 3 columns of strings
```

You can define variables define in the main program, or in user-defined functions. Those defined in the main program are global and can be accessed from anywhere in the script after their definition. Variables defined in user-defined functions exist from the point of definition to the end of the function and are deleted when the function ends. If you have a recursive function, each time you enter the function you get a fresh set of variables.

The dimensions of global arrays must be constant expressions. The dimensions of local arrays can be set by variables or calculated expressions. Simple variables (not arrays) can be initialised to constants when they are declared. The initialising expression may not include variables or function calls. Uninitialised numeric variables are set to 0, uninitialised strings are empty.

```
var Sam%:=3+2, jim := 2.3214, sally$ := "My name is \"Sally\"";
```

Constant declarations

Constants are created by the `const` keyword. A constant can be of any of the three basic data types, and must be initialised as part of the constant declaration. Constants cannot be arrays. The syntax and use of constants is the same as for variables, except that you cannot assign to them or pass them to a function or procedure as a reference parameter.

```
const Sam%:=3+2, jim := 2.3214, sally$ := "My name is \"Sally\"";
```

Expressions and operators

Anywhere in the script where a numeric value can be used, so can a numeric expression. Anywhere a string constant can be used, so can a string expression. Expressions are formed from functions, variables, constants, brackets and operators. In numerical expressions, the following operators are allowed, listed in order of precedence:

Numeric operators

	Operators	**Names**
Highest	`` ` ``, `[]`, `()`	Matrix transpose, subscript, round brackets
	`-`, `not`	Unary minus, logical not
	`*`, `/`, `mod`	Multiply, divide and modulus (remainder)
	`+`, `-`	Add and subtract
	`<`, `<=`, `>`, `>=`	Less, less or equal, greater, greater or equal
	`=`, `<>`	Equal and not equal
	`and`, `band`	Logical `and`, bitwise `and`
	`or`, `xor`, `bor`, `bxor`	Logical `or`, exclusive `or` and bitwise versions
Lowest	`?:`	Ternary operator

The order of precedence determines the order in which operators are applied within an expression. Without rules on the order of precedence, `4+2*3` could be interpreted as 18 or 10 depending on whether the add or the multiply was done first. Our rules say that multiply has a higher precedence, so the result is 10. If in doubt, use round brackets, as in `4+(2*3)` to make your meaning clear. Extra brackets do not slow down the script.

The divide operator returns an integer result if both the divisor and the dividend are integers. If either is a real value, the result is a real. So `1/3` evaluates to `0`, while `1.0/3`, `1/3.0` and `1.0/3.0` all evaluate to `0.333333…`

The minus sign occurs twice in the list because minus is used in two distinct ways: to form the difference of two values (as in `fred:=23-jim`) and to negate a single value (`fred :=-jim`). Operators that work with two values are called *binary*, operators that work with a single value are called *unary*. There are four unary operators, `[]`, `()`, `-` and `not`, the remainder are binary.

There is no explicit `TRUE` or `FALSE` keyword in the language. The value zero is treated as false, and any non-zero value is treated as true. Logical comparisons have the value 1 for true. So `not 0` has the value `1`, and the `not` of any other value is `0`. If you use a real number for a logical test, remember that the only way to guarantee that a real number is zero is by assigning zero to it. For example, the following loop may never end:

```
var add:=1.0;
repeat
   add := add - 1.0/3.0; ' beware, 1/3 would have the value 0!
until add = 0.0;         ' beware, add may never be exactly 0
```

Even changing the final test to `add<=0.0` leads to a loop that could cycle 3 or 4 times depending on the implementation of floating point numbers.

The result of the comparison operators is integer 0 if the comparison is false and integer 1 if the comparison is true. The result of the binary arithmetic operators is integer if both operands are integers, otherwise the result is a real number. The result of the logical operators is integer 0 or 1. The result of the exclusive or operator is true if one operand is true and the other is false.

The bitwise operators `band`, `bor` and `bxor` treat their operands as integers, and produce an integer result on a bit by bit basis. They are not allowed with real number operands.

String operators

	Operators	**Names**
Highest	`+`	Concatenate
	`<, <=, >, >=`	Less, less or equal, greater, greater or equal
	`=, <>`	Equal and not equal
Lowest	`?:`	Ternary operator

The comparison operators can be applied to strings. Strings are compared character by character, from left to right. The comparison is case sensitive. To be equal, two strings must be identical. You can also use the + operator with strings to concatenate them (join them together). The character order for comparisons (lowest to highest) is:

```
space !"#$%&'()*+,-./0123456789:;<=>?@
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`
abcdefghijklmnopqrstuvwxyz{|}~
```

Do not confuse assignment := with the equality comparison operator, =. They are entirely different. The result of an assignment does not have a value, so you cannot write statements like `a:=b:=c;`.

Examples of expressions

The following (meaningless) code gives examples of expressions.

```
var jim,fred,sam,sue%,pip%,alf$,jane$;
jim := Sin(0.25) + Cos(0.25);
fred := 2 + 3 * 4;      'Result is 14.0 as * has higher precedence
fred := (2 + 3)* 4;     'Result is 20.0
fred += 1;              'Add 1 to fred
sue% := 49.734;         'Result is 49
sue% := -49.734;        'Result is -49
pip% := 1 + fred > 9;   'Result is 1 as 21.0 is greater than 9
jane$ := pip% > 0 ? "Jane" : "John"; 'Result is "Jane"
alf$ := "alf";
sam := jane$ > alf$;    'Result is 0.0 (a is greater than J)
sam := UCase$(jane$)>UCase$(alf$); 'Result is 1.0 (A < J)
sam := "same" > "sam";  'Result is 1.0
pip% := 23 mod 7;       'Result is 2
jim := 23 mod 6.5;      'Result is 3.5
jim := -32 mod 6;       'Result is -2.0
sue% := jim and not sam;'Result is 0 (jim = -2.0 and sam = 1.0)
pip% := 1 and 0 or 2>1; 'Result is 1
sue% := 9 band 8;       'Result is 8 (9=1001 in binary, 8=1000)
sue% := 9 bxor 8;       'Result is 1
sue% := 9 bor 8;        'Result is 9
```

Mathematical constants

We don't provide maths constants as built-in symbols, but the two most common ones, *e* and π are easily generated within a script; *e* is `Exp(1.0)` and π is `4.0*ATan(1.0)`.

Flow of control statements

If scripts were simply a list of commands to be executed in order, their usefulness would be severely limited. The flow of control statements let scripts loop and branch. It is considered good practice to keep flow of control statements on separate lines, but the script syntax does not require this. There are two branching statements, `if...endif` and `docase...endcase`, and three looping statements, `repeat...until`, `while...wend` and `for...next`. You can also use user-defined functions and procedures with the `Func` and `Proc` statements.

if...endif

The `if` statement can be used in two ways. When used without an `else`, a single section of code can be executed conditionally. When used with an `else`, one of two sections of code is executed. If you need more than two alternative branches, the `docase` statement is usually more compact than nesting many `if` statements.

```
if expression then              'The simple form of an if
   zero or more statements;
endif;

if expression then              'Using an else
   zero or more statements;
else
   zero or more statements;
endif;
```

If the expression is non-zero, the statements after the `then` are executed. If the expression is zero, only the statements after the `else` are executed. The following code adds 1 or 2 to a number, depending on it being odd or even:

```
if num% mod 2 then
   num%:=num%+2;           'note that the semicolons before...
else                       '...the else and endif are optional.
   num%:=num%+1;
endif;
                           'The following is equivalent
if num% mod 2 then num%:=num%+2 else num%:=num%+1 endif;
```

docase...endcase These keywords enclose a list of `case` statements forming a multiway branch. Each `case` is scanned until one is found with a non-zero expression, or the `else` is found. If the `else` is omitted, control passes to the statement after the `endcase` if no case expression is non-zero. Only the first non-zero `case` is executed (or the `else` if no case is non-zero).

```
docase
 case exp1 then
   statement list;
 case exp2 then
   statement list;
 ...
 else
   statement list;
endcase;
```

The following example sets a string value depending on the value of a number:

```
var base%:=8,msg$;
docase
 case base%=2 then  msg$ := "Binary";
 case base%=8 then  msg$ := "Octal";
 case base%=10 then msg$ := "Decimal";
 case base%=16 then msg$ := "Hexadecimal";
 else msg$ := "Pardon?";
endcase;
```

repeat...until The statements between `repeat` and `until` are repeated until the expression after the `until` keyword evaluates to non-zero. The body of a repeat loop is always executed at least once. If you need the possibility of zero executions, use a `while` loop. The syntax of the statement is:

```
repeat
   zero or more statements;
until expression;
```

For example, the following code prints the times of all data items on channel 3 (plus an extra -1 at the end):

```
var time := -1;                  'start time of search
repeat
   time := NextTime(3, time);    'find next item time
   PrintLog("%f\n", time);       'display the time to the log
until time<0;                    'until no data found
```

while...wend The statements between the keywords are repeated while an expression is not zero. If the expression is zero the first time, the statements in between are not executed. The `while` loop can be executed zero times, unlike the `repeat` loop, which is always executed at least once.

```
while expression do
   zero or more statements;
wend;
```

The following code fragment, finds the first number that is a power of two that is greater than or equal to some number:

```
var test%:=437, try%:=1;
while try%<test% do              'if try% is too small...
   try% := try% * 2;             '...double it
wend;
```

for...next A `for` loop executes a group of statements a number of times with a variable changed by a fixed amount on each iteration. The loop can be executed zero times. The syntax is:

```
for v := exp1 to exp2 {step exp3} do
   zero or more statements;
next;
```

`v` This is the loop variable and may be a real number, or an integer. It must be a simple variable, not an array element.

`exp1` This expression sets the initial variable value before the looping begins.

`exp2` This expression is evaluated once, before the loop starts, and is used to test for the end of the loop. If `step` is positive or omitted (when it is 1), the loop stops when the variable is greater than `exp2`. If `step` is negative, the loop stops when the variable is less than `exp2`.

`exp3` This expression is evaluated once, before the loop starts, and sets the increment added to the variable when the `next` statement is reached. If there is no `step exp3` in the code, the increment is 1. The value of `exp3` can be positive or negative.

The following example prints the squares of all the integers between 10 and 1:

```
var num%;
for num% := 10 to 1 step -1 do
   PrintLog("%d squared is %d\n", num%, num% * num%);
next;
```

If you want a `for` loop where the end value and/or the `step` size are evaluated each time round the loop you should use a `while…wend` or `repeat…until` construction.

Halt The `Halt` keyword terminates a script. A script also terminates if the path of execution reaches the end of the script. When a script halts, any open external files associated with the `Read()` or `Print()` functions are closed and any windows with invalid regions are updated. Control then returns to the user.

Functions and procedures

A user-defined function is a named block of code. It can access global variables and create its own local variables. Information is passed into user-defined functions through arguments. Information can be returned by giving the function a value, by altering the values of arguments passed by reference or by changing global variables.

User-defined functions that return a value are introduced by the `func` keyword, those that do not are introduced by the `proc` keyword. The `end` keyword marks the end of a function. The `return` keyword returns from a function. If `return` is omitted, the function returns to the caller when it reaches the `end` statement. Arguments can be passed to functions by enclosing them in brackets after the function. Functions that return a value or a string have names that specify the type of the returned value. A function is defined as:

```
func name({argument list})              or     proc name({argument list})
{var local-variable-list;}                     {var local-variable-list;}
statements including return x;                 statements including return;
end;                                           end;
```

There is no semicolon at the end of the argument list because the argument list is not the end of the `func` or `proc` statement; the `end` keyword terminates the statement. Functions may not be nested within each other.

Argument lists The argument list is a list of variable names separated by commas. There are two ways to pass arguments to a function: by value and by reference:

Value Arguments passed by value are local variables in the function. Their initial values are passed from the calling context. Changes made in the function to a variable passed by value do not affect the calling context.

Reference Arguments passed by reference are the same variables (by a different name) as the variables passed from the calling context. Changes made to arguments passed by reference do affect the calling context. Because of this, reference arguments must be passed as variables (not expressions or constants) and the variable must match the type of the argument (except we allow you to pass a real variable where an integer variable is expected).

Simple (non-array) variables are passed by value. Simple variables can be passed by reference by placing the `&` character before the name in the argument list declaration.

Arrays and sub-arrays are always passed by reference. Array arguments have empty square brackets in the function declaration, for example `one[]` for a vector and `two[][]` for a matrix. The number of dimensions of the passed array must match the declaration. The array passed in sets the size of each dimension. You can find the size of an array with the `Len()` function. An individual array element is a treated as a simple variable.

If you use the `trans()` or `diag()` operators to pass the transpose or diagonal of an array to a function, the array is still passed by reference and changes made in the function to array elements will change the corresponding elements in the original data.

return The `return` keyword is used in a user-defined function to return control to the calling context. In a `proc`, the `return` must not be followed by a value. In a `func`, the return should be followed by a value of the correct type to match the function name. If no return value is specified, a `func` that returns a real or integer value returns 0, and a `func` that returns a string value returns a string of zero length.

Examples of user-defined functions

```
proc PrintInfo()            'no return value, no arguments
PrintLog(ChanTitle$(1));    'Show the channel title
return;                     'return is optional in this case as...
end;                        '...end forces a return for a proc

func sumsq(a, b)            'sum the square of the arguments
return a*a + b*b;
end;

func removeExt$(name$)      'remove text after last . in a string
var n := 0, k := 1;
   repeat
   k:=InStr(name$,".",k);   'find position of next dot
   if (k > 0) then          'if found a new dot...
      n := k;               '...remember where
   endif
   until k=0;               'until all found
if n=0 then
   return name$;            'no extension
else
   return Left$(name$,n-1);
end;
```

```
proc sumdiff(&arg1, &arg2) 'returns sum and difference of args
arg1 := arg1 + arg2;       'sum of arguments
arg2 := arg1 - 2*arg2;     'original arg1-arg2
return;                    'results returned via arguments
end;

func sumArr(data[])        'sum all elements of a vector
var sum:=0.0;              'initialise the total
var i%;                    'index
for i%:=0 to Len(data[])-1 do
   sum := sum + data[i%];  'of course, ArrSum() is much faster!
   next;
return sum;
end;

Func SumArr2(data[][])      'Sum of all matrix elements
var rows%,cols%,i%,sum;     'sizes, index and sum, all set to 0
rows% := Len(data[0][])     'get sizes of dimensions...
cols% := Len(data[][0]);    '...so we can see which is bigger
if rows%>cols% then         'choose more efficient method
 for i%:=0 to cols%-1 do sum += ArrSum(data[i%][]) next;
else
 for i%:=0 to rows%-1 do sum += ArrSum(data[][i%]) next;
endif;
return sum;
end;
```

Variables declared within a function exist only within the body of the function. They cannot be used from elsewhere. You can use the same name for variables in different functions. Each variable is separate. In addition, if you call a function recursively (that is it calls itself), each time you enter the function, you have a fresh set of variables.

Scope of user-defined functions

Unlike global variables, which are only visible from the point in the script in which they are declared onwards, and local variables, which are visible within a user-defined function only, user-defined functions are visible from all points in the script. You may define two functions that call each other, if you wish.

Functions as arguments

The script language allows a function or procedure to be passed as an argument. The function declaration includes the declaration of the function type to be passed. Functions and procedures can occur before or after the line in which they are used as an argument.

```
proc Sam(a,b$,c%)
...
end;

func Calc(va)
return 3*va*va-2.0*va;
end;

func PassFunc(x, func ff(realarg))
return ff(x);
end;

func PassProc(proc jim(realarg, strArg$, son%))
jim(1.0,"hello",3);
end;

val := PassFunc(1.0, Calc);  'pass function
PassProc( Sam );             'pass procedure
```

The declaration of the procedure or function argument is exactly the same as for declaring a user-defined function or procedure. When passing the function or procedure as an argument, just give the name of the function or procedure; no brackets or arguments are required. The compiler checks that the argument types of a function passed as an argument match those declared in the function header. See the `ToolbarSet()` function for an example.

Although user-defined functions and built-in functions are very similar, you are not allowed to pass a built-in function as an argument to a built-in function. Further, you cannot pass a built-in function to a user-defined function if it has one or more arguments that can be of more than one type. For example, the built-in `Sin()` function can accept a real argument, or a real array argument, and so cannot be passed.

Channel specifiers

Several built-in script commands use a channel specifier to define a list of 1 or more channels. This argument is always called `cSpc` in the documentation. This argument stands for a string, an integer array or an integer.

`cSpc$` This holds a list of channel numbers and channel ranges, separated by commas. A channel range is a start channel number followed by an end channel number separated by two dots. The end channel number can be less that the start channel number. For example `"1,3,5..7,12..9"` is a list of channels 1, 3, 5, 6, 7, 9, 10, 11 and 12.

`cSpc%[]` The first element of the vector is the number of channels. The remaining elements are channel numbers. It is an error for the vector passed in to be shorter than the number of channels + 1.

`cSpc%` This is either a channel number, or –1 for all channels, -2 for visible channels or –3 for selected channels.

Some commands expect channels of specific types; channels that do not meet the type requirements are removed from the list. It is usually an error for a channel specification to generate an empty list.

4 Commands by function

Functional command groups

This section of the manual lists commands by function. The next section lists the command alphabetically with a description of the command arguments and operation.

Windows and views

These commands are used to manipulate windows (views) to position them, display and size them, colour them and create them.

`App`	Get the application window view handle
`ChanNumbers`	Show or hide channel numbers
`Colour`	Get or set the palette entry associated with a screen item
`Dup`	Get the view handle of one of the duplicates of the current view
`Draw`	Draw invalid regions of the view (and set x axis range)
`DrawAll`	Update all invalid regions in all views
`FileClose`	Closes a window or windows
`FileComment$`	Gets and sets the file comment for time views
`FileConvert$`	Convert a foreign file to a Signal file and open it
`FileName$`	Gets the file name associated with a window
`FileNew`	Opens an output file or a new text or data window
`FileOpen`	Opens an existing file (in a window)
`FilePrint`	Prints a range of data from the current view
`FilePrintVisible`	Prints the current view
`FilePrintScreen`	Prints all text-based and data views
`FileQuit`	Closes the Signal application
`FileExportAs`	Export from a data view in a variety of formats
`FileSave`	Save a view with same name
`FileSaveAs`	Save a view with specified new name
`FocusHandle`	Get the view handle of the window which has the input focus
`FontGet`	Read back information about the font
`FontSet`	Set the font for the current window
`FrontView`	Get or set the front window on screen
`Grid`	Get or set the visibility of the axis grid
`LogHandle`	Gets the view handle of the log window
`PaletteGet`	Get the RGB colour of a palette entry
`PaletteSet`	Set the RGB colour of a palette entry
`SampleHandle`	Gets the view handle of sampling windows and controls
`ViewColour`	Override application colours for a view
`View`	Change or override current view and get view handle
`ViewFind`	Get a view handle from a view title
`ViewKind`	Get the type of the current view or other view
`ViewList`	Form a list of handles of views that meet a specification
`ViewStandard`	Returns a window to a standard state
`ViewUseColour`	Get and set monochrome/colour use for view
`Window`	Sets the window size and position
`WindowDuplicate`	Duplicate a time view
`WindowGetPos`	Get window position
`WindowSize`	Changes the window size
`WindowTitle$`	Gets or changes the window title
`WindowVisible`	Sets or gets the visibility of the window (hide/show)
`XAxis`	Get or set the visibility of the x axis
`XAxisAttrib`	Get or set x axis attributes such as logarithmic
`XAxisMode`	Get or set the visibility of the x axis' features (eg large ticks)
`XAxisStyle`	Get or set the x axis major and minor tick spacing
`YAxis`	Get or set the visibility of the y axis
`YAxisAttrib`	Get or set y axis attributes such as logarithmic
`YAxisMode`	Get or set the visibility of the y axis' features (eg large ticks)
`YAxisStyle`	Get or set the y axis major and minor tick spacing

Data views

These commands operate on any data view, whether a file view, memory view or sampling document view.

`AppendFrame`	Add a new data frame to end of data
`DeleteFrame`	Delete a data frame if not on disk
`ExportChanFormat`	Set channel format for export
`ExportChanList`	Set list of channels for export
`ExportFrameList`	Set a list of frames for export
`ExportTextFormat`	Set format for text output of channels
`ExportTimeRange`	Set x axis range for export of data
`FileExportAs`	Export from a data view in a variety of formats
`Frame`	Get or set the current frame
`FrameAbsStart`	Get or set the current frame absolute start time
`FrameComment$`	Get or set the comment with the current frame
`FrameCount`	The number of frames in the file or memory view
`FrameFlag`	Gets or sets a frame flag
`FrameList`	Fills an array with frame numbers according to a frame spec
`FrameMean`	Gets or sets the flag saying if the frame shows a mean or a total
`FrameSave`	Saves changed frame data or discards changes
`FrameState`	Gets or sets the frame state value
`FrameTag`	Gets or sets the frame tag
`FrameUserVar`	Gets or sets a frame user variable
`Maxtime`	Maximum x axis value in the current frame
`Mintime`	Minimum x axis value in the current frame
`Overdraw`	Enables or disables drawing the frame display list
`OverdrawFrames`	Sets a frame display list for the view
`OverdrawGetFrames`	Gets the frame numbers from the display list for the view
`ShowBuffer`	Get or set the frame buffer display state
`Sweeps`	Number of items processed into memory view
`TimeRatio`	Get the scaling factor from X units to seconds
`TimeUnits$`	Get the current time units
`XLow`	The start of the displayed area in x axis units
`XHigh`	The end of the displayed area in x axis units
`XRange`	Set the x axis range for next draw
`XScroller`	Show or hide the x axis scroll bar and controls
`XTitle$`	X axis title. Can be set, in a memory or sampling document view
`XUnits$`	X axis units. Can be set, in a memory or sampling document view

Vertical cursors

The following commands control the vertical cursors. Where possible, changes to cursors cause immediate screen changes; changes do not wait for the next `Draw` command. This is unlike almost all other commands, which save up changes until the next draw.

`Cursor`	Set or get the position of a cursor
`CursorActiveGet`	Get the active cursor parameters
`CursorActiveSet`	Set the active cursor parameters
`CursorDelete`	Delete a designated cursor
`CursorExists`	Test if a cursor exists
`CursorLabel`	Set or get the cursor label style
`CursorLabelPos`	Set or get the cursor label position
`CursorMode`	Set or get the cursor mode
`CursorNew`	Add a new cursor (at a given position)
`CursorRenumber`	Renumber the cursors in ascending position order
`CursorSet`	Set the number (and position) of vertical cursors

Horizontal cursors

Horizontal cursors belong to a channel, but can be dragged to different channels within a view by the user. Horizontal cursors have a value and are drawn at the y axis position corresponding to the value. If the value is beyond the range of the y axis, the cursor is invisible. If you delete a channel with horizontal cursors, the cursors are deleted.

`HCursor`	Set or get the position of a horizontal cursor
`HCursorChan`	Gets the channel that a horizontal cursor belongs to
`HCursorDelete`	Delete a designated horizontal cursor
`HCursorExists`	Test if a horizontal cursor exists
`HCursorLabel`	Gets or sets the horizontal cursor style
`HCursorLabelPos`	Gets or sets the horizontal cursor position
`HCursorNew`	Add a new horizontal cursor on a channel (at a given position)
`HCursorRenumber`	Renumbers the cursors from bottom to the top of the view

Channels

These commands operate on channels in a data or XY view. Channel data can also be treated as an array, so you can use all the array arithmetic commands. In a memory view, you can use the commands `BinSize` and `BinZero` to get or set x axis scale and offset, but in any other data view you can use these commands only to get x axis values.

`BinError`	Error value for a given bin
`BinSize`	X axis interval for waveforms or resolution for markers
`BinToX`	Bin or item number at x axis position
`BinZero`	X axis position of the first bin on the channel
`Chan$`	Converts a channel number or list into a string
`ChanCount`	Count channels of a given type
`ChanDelete`	Delete a channel from an XY view or an idealised trace
`ChanDiff`	Differentiate data in specified channels
`ChanIntgl`	Integrate data in specified channels
`ChanItems`	Count items in a channel within an x axis range
`ChanKind`	Get the type of a channel
`ChanList`	Get a list of channels meeting a specification
`ChanMean`	Mean of waveform level within an x axis range
`ChanMeasure`	Take a variety of measurements on channel data
`ChanNegate`	Negate (invert) data in specified channels
`ChanOffset`	Offset data in specified channels by a constant value
`ChanOrder`	Modify the channel order and y axis grouping
`ChanPoints`	Number of data items in the channel in the frame
`ChanRectify`	Rectify data in specified channels
`ChanScale`	Scale data in specified channels by a constant value
`ChanSearch`	Scan a channel of data for a particular feature
`ChanSelect`	Select and report on selected state of channels
`ChanShift`	Shift data in specified channels left or right
`ChanShow`	Make a channel visible or invisible
`ChanSmooth`	Smooth (3 or 5 point) data in specified channels
`ChanSubDC`	Remove DC offset from data in specified channels
`ChanRange`	Get start and count of items in an x axis range
`ChanTitle$`	Get or set the channel title string
`ChanUnits$`	Get or set the channel units
`ChanValue`	Get channel data at a particular time or x axis position
`ChanVisible`	Get the visibility state of a channel
`ChanWeight`	Change the relative vertical space of a channel
`ChanZero`	Clear data in specified channels
`DrawMode`	Get or set display mode for a channel
`LastTime`	Find the previous item in a channel (and return values)
`MarkCode`	Get a marker code(s)

`MarkEdit`	Edit a marker code(s)
`MarkTime`	Change position of a marker
`Maxtime`	Time of last item on the channel
`Minmax`	Find minimum and maximum values (and positions)
`Mintime`	Time of first item on the channel
`NextTime`	Find the next item in a channel (and return values)
`Optimise`	Set reasonable y range for channels with axes
`XToBin`	Convert x axis value to bin number
`YLow`	Get lower limit of y axis for a channel
`YHigh`	Get upper limit of y axis for a channel
`YRange`	Set y axis range for a channel

Buffer

These commands operate on the frame buffer that is attached to each file or memory view. The buffer can be shown or hidden using the `ShowBuffer` function. These commands perform arithmetic between the buffer and a frame in the data view. These functions operate on all points in all waveform channels in the buffer. The functions that modify a frame in the data document have names such as `BuffAddTo` or `BuffMulBy`, while functions that change the frame buffer are called `BuffAdd` or `BuffMul`. Note that you can use the channel data manipulation commands to change buffer data, as well as accessing the buffer data directly.

`BuffAdd`	Add data to the frame buffer
`BuffAddTo`	Add frame buffer to a data framer
`BuffAcc`	Add data to average in frame buffer
`BuffClear`	Clear all channels in the frame buffer
`BuffCopy`	Copy data to the frame buffer
`BuffCopyTo`	Copy frame buffer to the data frame
`BuffDiv`	Divide frame buffer by data
`BuffDivBy`	Divide data frame by the frame buffer
`BuffExchange`	Exchange data in the frame buffer and data frame
`BuffMul`	Multiply frame buffer by data
`BuffMulBy`	Multiply data frame by frame buffer
`BuffSub`	Subtract data from frame buffer
`BuffSubFrom`	Subtract frame buffer from a data frame
`BuffUnAcc`	Remove data from average in frame buffer

XY views

These commands specifically manipulate XY views. XY views have from 1 to 100 channels of data. Each channel holds a list of (x, y) co-ordinate pairs that can be displayed in a variety of styles. Most functions that work on views in general will also work on an XY view.

`XYAddData`	Add data points to a channel of an XY view
`XYColour`	Set the colour of a channel
`XYCount`	Return the number of XY data points in a channel
`XYDelete`	Delete one or more data points from a channel
`XYDrawMode`	Control how a channel is drawn
`XYGetData`	Read data back from an XY channel
`XYInCircle`	Count the number of XY points within a circle
`XYInRect`	Count the number of XY points inside a rectangle
`XYJoin`	Get or set the point joining method
`XYKey`	Control the display of the XY view channel key
`XYRange`	Get rectangle containing one or more channels
`XYSetChan`	Create or modify an XY channel
`XYSize`	Get or set maximum size of an XY channel
`XYSort`	Change the sort (and draw) order of a channel

Sampling configuration commands

These commands correspond to actions on the Sampling configuration dialog. They get or change the sampling configuration settings that will be used the next time you create a new Signal data file for sampling.

These commands correspond to the General page of the configuration, or are general-purpose in intent. A few of these can also interact with sampling in progress.

`SampleBurst`	Set or get the burst mode flag
`SampleClear`	Set the Sampling configuration to a known state
`SampleDigMark`	Add or remove the digital marker channel
`SampleKeyMark`	Add or remove the keyboard marker channel
`SampleMode`	Set or get the sweep mode for sampling
`SamplePause`	Set or get pause at sweep end flag
`SamplePoints`	Set or get the number of data points per ADC port
`SamplePorts`	Set or get which ADC ports to sample from
`SampleRate`	Set or get the ADC sample rate per channel in Hz
`SampleTrigger`	Set or get the triggered sweeps flag
`SampleWrite`	Control writing data to sampling file
`SampleZeroOffset`	Set or get the x-axis zero offset.

These commands correspond to the Ports page of the sampling configuration.

`SamplePortFull`	Set ADC port value for full input
`SamplePortName$`	Set ADC port title
`SamplePortOptions$`	Set ADC port online processing options
`SamplePortUnits$`	Set ADC port units
`SamplePortZero`	Set ADC port value for zero on the input
`SampleTel`	Set up of telegraph gains for an amplifier

These commands correspond to the Outputs page of the sampling configuration.

`SampleDacFull`	Set or get a DAC full-scale value
`SampleDacMask`	Set or get the DAC output enable mask
`SampleDacUnits$`	Set or get a DAC units string
`SampleDacZero`	Set or get a DAC zero value
`SampleDigOMask`	Set or get the digital outputs enable mask
`SampleOutClock`	Set or get the outputs clock period
`SampleOutMode`	Set or get the outputs mode

These commands correspond to the States page of the sampling configuration.

`SampleAuxStateParam`	Set or get parameters for the auxiliary states device
`SampleAuxStateValue`	Set or get auxiliary states device settings
`SampleDigIMask`	Set or get digital inputs enable mask
`SampleStates`	Set or get states enable and number of extra states
`SampleStatesIdle`	Set or get states ordering cycles before idling
`SampleStatesMode`	Set or get multiple states mode
`SampleStatesOrder`	Set or get multiple states ordering mode
`SampleStatesRepeats`	Set or get multiple states repeats count
`SampleStateDac`	Set or get DAC data for individual state
`SampleStateDig`	Set or get digital bits for individual state
`SampleStateRepeats`	Set or get repeats for individual state

These correspond to the Protocols dialog available from the States page.

`Protocols`	Get number of protocols set up
`ProtocolAdd`	Add a new protocol to list
`ProtocolClear`	Initialise a protocol
`ProtocolDel`	Delete a protocol from list
`ProtocolEnd`	Set or get what happens when the end of a protocol is reached
`ProtocolFlags`	Set or get protocol flags
`ProtocolName$`	Set or get protocol name
`ProtocolStepGet`	Get information on protocol step
`ProtocolStepSet`	Set protocol step values

These commands correspond to the Automation page of the sampling configuration.

`SampleArtefactGet`	Get the artefact rejection settings
`SampleArtefactSet`	Set the artefact rejection settings
`SampleAutoFile`	Set or get the automatic file save flag
`SampleAutoName$`	Set or get the template for automatic file naming
`SampleLimitFrames`	Set or clear the limit on the number of frames in the new file
`SampleLimitSize`	Set or clear the size limit of the new file
`SampleLimitTime`	Set or clear the limit on the overall sampling time

These commands correspond to the Peri-trigger page of the sampling configuration.

`SamplePeriDigBit`	Set digital bit for peri-trigger digital type
`SamplePeriBitState`	Set digital triggering to be on bit high or low
`SamplePeriHyst`	Set hysteresis value for peri-trigger + or - analog type
`SamplePeriLevel`	Set threshold level for peri-trigger analog types
`SamplePeriLowLev`	Set lower threshold level for peri-trigger =analog type
`SamplePeriType`	Set type of peri-trigger
`SamplePeriPoints`	Set peri-trigger pre-trigger points

These commands correspond to the pulse dialog or to outputs page items that specifically interact with the pulses output. The pulse functions can all be used while sampling is in progress to alter the pulses in use.

`SampleAbsLevel`	Set or get the pulses absolute levels flag (in outputs page)
`SampleFixedInt`	Set or get the fixed interval sweep mode interval
`SampleFixedVar`	Set or get the fixed interval percentage variation
`SampleOutLength`	Set or get the pulses output frame length
`SampleOutTrig`	Set or get the pulses sampling sweep trigger time
`Pulses`	Get the number of pulses for an output port
`PulseAdd`	Add a new pulse to the outputs for a port
`PulseClear`	Remove all pulses from the outputs for a port
`PulseDel`	Remove a pulse from the outputs for a port
`PulseFlags`	Set or get the options flags for a pulse
`PulseName$`	Set or get a pulse name
`PulseType`	Get a pulse type code
`PulseDataSet`	Set the amplitude and other values for a pulse
`PulseDataGet`	Get the amplitude and other values for a pulse
`PulseVarSet`	Set the variation parameters for a pulse
`PulseVarSet`	Get the variation parameters for a pulse
`PulseTimesSet`	Set the times for a pulse
`PulseTimesGet`	Get the times for a pulse
`PulseWaveSet`	Set the waveform output parameters
`PulseWaveGet`	Get the waveform output parameters
`PulseWaveformSet`	Set the waveform output data for a DAC
`PulseWaveformGet`	Get the waveform output data for a DAC

These commands correspond to the outputs sequencer.

`SampleSeqCtrl`	Sets where a sequencer can be controlled from
`SampleSeqStep`	Gets the current sequencer step
`SampleSeqTable`	Gets the size of any table set in the sequencer
`SampleSequencer`	Sets the sequencer file to use
`SampleSequencer$`	Gets the sequencer file name in use
`SampleSeqVar`	Sets variable values used in the sequencer
`SampleSeqWave`	Sets up waveform output for use from the sequencer

Runtime sampling commands

These commands control and interact with the data sampling process. Signal samples into one data file at a time, and these commands refer to it, regardless of the current view. The commands can also be used to retrieve the current settings.

`SampleAbort`	Exit from sampling and throw data away
`SampleAccept`	Accept or reject the current sweep
`SampleHandle`	Gets the view handle of sampling windows and controls
`SampleKey`	Adds to the keyboard marker channel, controls output sequencer
`SamplePeriHyst`	Alter hysteresis for peri-trigger + or - analog types
`SamplePeriLevel`	Alter threshold level for peri-trigger analog types
`SamplePeriLowLev`	Alter lower threshold level for peri-trigger =analog type
`SampleProtocol`	Set protocol to be used for state sequencing
`SampleReset`	Clear all data from the new file and get ready to start again
`SampleStart`	Start sampling after creating a new time view
`SampleState`	Set state for next frame to be sampled a new time view
`SampleStatesPause`	Set or get the current sample state sequencer pausing
`SampleStatesReset`	Reset states sequencing and pulse variations
`SampleStatesRun`	Set state sequencing run mode or manual
`SampleStatus`	Get the current sampling state
`SampleStop`	Stop sampling and keep the data
`SampleSweep`	Start another sampling sweep

Analysis

These functions create new memory or XY views and define an analysis process for them.

`SetAmplitude`	Set up an amplitude histogram view derived from a file view
`SetAutoAv`	Set up an auto-average multi-frame view derived from a file view
`SetAverage`	Set up a waveform average view derived from a file view
`SetLeak`	Set up a leak-subtracted view produced from a file view
`SetOpCl`	Set up an idealised trace using threhold crossing
`SetOpClScan`	Set up an idealised trace using the scan method
`SetOpClAmp`	Set up an amplitude histogram formed from an idealised trace
`SetOpClBurst`	Set up a burst time histogram formed from an idealised trace
`SetOpClHist`	Set up a dwell time histogram from an idealised trace
`SetPower`	Set up a power spectrum view derived from a file view
`SetTrend`	Set up an XY trend plot derived from a file or memory view
`SetTrendChan`	Add a channel to XY trend plot set up by `SetTrend`

These functions create new memory views without an attached analysis process.

`SetCopy`	Set up a memory view copied from a file view
`SetMemory`	Set up a memory view for user-defined data

The Process commands work with views with an attached analysis process. They carry out the analysis process defined when setting up the memory view, processing data from the source view into the memory or XY view attached to it.

`Process`	Carry out the analysis process on the current frame from file
`ProcessAll`	Process all memory views attached to a file view
`ProcessFrames`	Carry out the analysis process on multiple frames from file
`ProcessOnline`	Carry out the analysis process during sampling
`Sweeps`	Number of items processed into memory view

Signal conditioner control

These functions control serial-line controlled signal conditioners.

`CondFilter`	Get or set the conditioner low-pass or high-pass filter
`CondFilterList`	Get a list of possible low-pass or high-pass filter settings
`CondGain`	Get or set the conditioner gain
`CondGainList`	Get a list of the possible gains for the conditioner
`CondGet`	Get all the settings for one channel of the conditioner
`CondOffset`	Get or set the conditioner offset for a channel
`CondOffsetLimit`	Get or set the conditioner offset range for a channel
`CondRevision$`	Get or set the conditioner offset for a channel
`CondSet`	Single call to set all channel parameters
`CondSourceList`	Get names of the signal sources available on the conditioner
`CondType`	Get the type of signal conditioner

Editing operations

These functions mimic the Edit menu commands and provide additional functionality.

`EditClear`	Delete text from a text window at the caret
`EditCopy`	Copy the current selection to the clipboard
`EditCut`	Delete the current selection to the clipboard
`EditPaste`	Paste the clipboard into the current text field
`EditSelectAll`	Select the entire text or cursor window contents
`MoveBy`	Move relative to current position
`MoveTo`	Move to a particular place
`OpClEventChop`	Splits an event from an idealised trace in two
`OpClEventDelete`	Deletes an event from an idealised trace
`OpClEventGet`	Gets the details of an event in an idealised trace
`OpClEventMerge`	Merges two events from an idealised trace
`OpClEventSet`	Sets the details of an event in an idealised trace
`OpClEventSplit`	Splits an event in an idealised trace into three
`Selection$`	This function returns the text that is currently selected

String functions

The following functions are used to manipulate strings and to convert between strings and other variable types.

`Asc`	ASCII code of first character of a string
`Chr$`	Converts a code to a one character string
`DelStr$`	Returns a string minus a substring
`InStr`	Searches for a string in another string
`LCase$`	Returns lower case version of a string
`Left$`	Returns the leftmost characters of a string
`Len`	Returns the length of a string or array
`Mid$`	Returns a substring of a string
`Print$`	Produce formatted string from variables
`ReadStr`	Extract variables from a string
`Right$`	Returns the rightmost characters of a string
`Str$`	Converts a number to a string
`UCase$`	Returns upper case version of a string
`Val`	Converts a string to number

Array and matrix arithmetic

These functions can be used with arrays and channel data to speed up data manipulation. In this section, the word "array" can be applied to an array declared with the `var` or `proc` or `func` statements, or to channel data in a file or memory view. Integer arrays can be used where indicated, but beware of overflow.

The functions all return a negative error code if there is a problem or zero if the function completes without error. The array arithmetic attempts to fix problems by setting the result element to a (possibly) useful value.

You can apply built-in mathematical functions directly on an array. For example, to form the square root of all the elements of array `fred[]` use `Sqrt(fred[])`. To access data in channel `c` of view `v` use `View(v,c).[{aExp}]` in place of `fred[{aExp}]` where `aExp` is an optional expression to specify elements as described in the script language syntax. For example, to subtract channel 2 from channel 1 in view `v1%`, use `ArrSub(View(v1%,1)[],View(v1%,2).[])`.

`ArrAdd`	Adds an array or constant to an array
`ArrConst`	Copies an array, or sets an array to a constant value
`ArrDiff`	Replaces an array with an array of simple differences
`ArrDiv`	Divides an array by another array or a constant
`ArrDivR`	Divides array into another array or constant
`ArrDot`	Forms the dot product (sum of products) of two arrays
`ArrFFT`	Fourier transform and related operations
`ArrFilt`	Applies a FIR filter to an array
`ArrIntgl`	Integrates array; inverse of `ArrDiff()`
`ArrMul`	Multiples an array by another array or constant
`ArrSpline`	Uses cubic splining to change the sample interval of an array
`ArrSub`	Subtract constant from array, or difference of two arrays
`ArrSubR`	Subtract array from constant, or reversed difference of arrays
`ArrSum`	Sum, mean and standard deviation of an array
`Len`	Returns the length of a string or array
`MATInv`	Inverts a matrix
`MATMul`	Matrix multiplication
`MATSolve`	Solves a matrix equation
`MATTrans`	Transposes a matrix matrix (also see the `trans()` operator)
`PCA`	Principal component analysis

Mathematical functions

The following mathematical functions are built into Signal. You can apply most of the arithmetic functions to real arrays by passing an array or channel data to the function.

`Abs`	Absolute value of a number or array
`ATan`	Arc tangent of number or array
`Cos`	Cosine of a number or array
`Exp`	Exponential function of a number or array
`Frac`	Remove integral part of a number or array
`GammaP`	The incomplete gamma function
`GammaQ`	1 – GammaP()
`Ln`	Natural logarithm of a number or array
`LnGamma`	The natural logarithm of the gamma function
`Log`	Logarithm to base 10 of a number or array
`Max`	Finds maximum of array or variables
`Min`	Finds minimum of array or variables
`Pow`	Raise a number or an array to a power
`Rand`	Returns a pseudo-random number
`Round`	Round a real number to the nearest integral value
`Sin`	Sine of a number or array
`Sqrt`	Square root of a number or an array
`Tan`	Tangent of a number or array
`Trunc`	Remove fractional part of number or array

Digital filtering

These functions create and apply digital filters and manipulate the filter bank.

`ArrFilt`	Array arithmetic routine to apply FIR coefficients to an array
`FiltApply`	Apply a set of coefficients or a filter bank filter to a waveform
`FiltAtten`	Set the desired attenuation of a filter in the filter bank
`FiltCalc`	Force coefficient calculation of a filter in the filter bank
`FiltComment$`	Get or set comment for a filter in the filter bank
`FiltCreate`	Create a new filter definition in the filter bank
`FiltInfo`	Retrieve information about a filter in the filter bank
`FiltName$`	Get or set the name of a filter in the filter bank
`FiltRange`	Get the useful sampling rate range for a filter bank filter
`FIRMake`	Generate FIR filter coefficients in an array
`FIRQuick`	Generate FIR filter coefficients with desired attenuation
`FIRResponse`	Calculate frequency response of array of coefficients
`IIRBp`	Create and/or apply an IIR bandpass filter
`IIRBs`	Create and/or apply an IIR bandstop filter
`IIRHp`	Create and/or apply an IIR highpass filter
`IIRLp`	Create and/or apply an IIR lowpass filter
`IIRNotch`	Create and/or apply an IIR notch filter
`IIRReson`	Create and/or apply an IIR resonator filter

Fitting functions

The following fitting functions are built into Signal.

`ChanFit`	A higher level fitting routine to emulate the fit dialog
`ChanFitCoef`	Get or set a fit coefficient
`ChanFitShow`	Show or hide a fit to a particular channel
`ChanFitValue`	Get the value of a fitted function
`FitData`	Fit a selected function to arrays of x,y data points
`FitExp`	Fit multiple exponentials to arrays of x,y data points
`FitGauss`	Fit multiple gaussians to arrays of x,y data points
`FitLine`	Fit a straight line to waveform channel data
`FitLinear`	Fits $y = \mathbf{a}_0 f_0(x) + \mathbf{a}_1 f_1(x) + \mathbf{a}_2 f_2(x)$ ... to a set of x,y data points
`FitNLUser`	Fit a non-linear user-defined function to a set of data points
FitOpClRange	Fits an idealised trace using the filter characteristics
`FitPoly`	Fits $\mathbf{a}_0 + \mathbf{a}_1 x + \mathbf{a}_2 x^2 + \mathbf{a}_3 x^3$ ... to a set of x,y data points
`FitSin`	Fits $\mathbf{a}_0\mathrm{Sin}(\mathbf{a}_1 x+\mathbf{a}_2) + \mathbf{a}_3\mathrm{Sin}(\mathbf{a}_4 x+\mathbf{a}_5)$ + ... to a set of x,y data points
`OpClFitRange`	Fits an idealised patch clamp trace to observed data

User interaction commands

These commands allow you to give information to, or get information from the user. They also let the user interact with the data.

`Input`	Prompt user for a number in a defined range
`Input$`	Prompt user for a string with a list of acceptable characters
`Interact`	Allow user to interact with data
`Message`	Display a message in a box, wait for OK
`Print`	Formatted text output to a file or window
`PrintLog`	Formatted text output to the Log window
`Print$`	Formatted text output to a string
`Query`	Ask a user a question, wait for response

You can build simple dialogs, with a set of fields stacked vertically or you can build free-format dialogs (but with more work to define the positions of all the dialog fields):

`DlgCreate`	Start a dialog definition
`DlgChan`	Define a dialog entry as prompt and channel selection
`DlgCheck`	Define a dialog item as a check box
`DlgInteger`	Define a dialog entry as prompt and integer number input

`DlgLabel`	Define a dialog entry as prompt only
`DlgList`	Define a dialog entry as prompt and selection from a list
`DlgReal`	Define a dialog entry as prompt and real number input
`DlgShow`	Display the dialog, get values of fields
`DlgString`	Define a dialog entry as prompt and string input
`DlgText`	Define a fixed text string for the dialog

The toolbar commands define the buttons available to the user on the toolbar, and allow the user to interact with the data, and have access to script functionality.

`Toolbar`	Let the user interact with the toolbar
`ToolbarClear`	Remove all defined buttons from the toolbar
`ToolbarEnable`	Get or set the enable state of a toolbar button
`ToolbarSet`	Add a button (and associate a function with it)
`ToolbarText`	Display a message using the toolbar
`ToolbarVisible`	Get or set the visibility of the toolbar
`SampleBar`	Controls the sample bar buttons
`ScriptBar`	Controls the script bar buttons

File system

Signal can read information about files and directories and also change the current directory and delete or copy files.

`FileCopy`	Copies a file from one place to another
`FileDelete`	Delete one or more files
`FileList`	Get a list of files or directories
`FilePath$`	Get the current directory or directory for new data files
`FilePathSet`	Change the current directory or directory for new data files

Text files

Signal can create text files and read from them. You can also open a text file into a window.

`FileNew`	Open a new text file in a window
`FileOpen`	Open a text file in a window or for reading and writing
`FileSaveAs`	Save a view in a variety of formats, including text
`Print`	Write formatted output to a file or log window
`Read`	Extract data from a text file
`ReadSetup`	Specify what characters will delimit read text

CFS variables

The following script commands read file and frame variables from CFS files written by other software. For those familiar with the CFS library for programming in DOS, the frames were referred to as data sections (DS).

`FileGetIntVar`	Read the value of an integer file variable
`FileGetRealVar`	Read the value of a floating point file variable
`FileGetStrVar$`	Read a string file variable
`FileVarCount`	Get the number of file variables in the file
`FileVarInfo`	Get the type and name of a numbered file variable
`FrameGetIntVar`	Read the value of an integer frame variable
`FrameGetRealVar`	Read the value of a floating point frame variable
`FrameGetStrVar$`	Read a string frame variable
`FrameVarCount`	Get the number of frame variables in the file
`FrameVarInfo`	Get the type and name of a numbered frame variable

Binary files

Signal can read and write binary files. These provide links to other software and are generally more efficient than text for passing large quantities of data between programs.

`FileClose`	Close a file opened in binary mode
`FileOpen`	Open an external file in binary mode
`BRead`	Extract 32-bit integer, 64-bit real and string data from a file
`BReadSize`	Extract 8 and 16-bit integer and 32-bit real data from a file
`BRWEndian`	Change the byte order for data in the file
`BSeek`	Change the current file position for next read or write
`BWrite`	Write 32-bit integer and 64-bit real data to a file
`BWriteSize`	Write 8 and 16-bit integer, 32-bit real and string data to a file

Serial line control

These functions let the script writer read to and write from serial line ports on their computer. This feature can be used to control equipment during data capture, although they are not needed for controlling the signal conditioners for which the CondXXX family of commands are provided.

`SerialOpen`	Open a serial port and configure it (set Baud rate, parity etc.)
`SerialWrite`	Write characters to the serial port
`SerialRead`	Read characters from the serial port
`SerialCount`	Count the number of data items available to read
`SerialClose`	Release a previously opened serial port

Debugging operations

These functions can be used when debugging a script.

`Debug`	Set a permanent break point or disable the `Esc` key
`Eval`	Convert the argument to text and display

Environment

These functions don't fit well into any of the other categories!

`Date$`	Get system date in a string in a variety of formats
`Error$`	Convert a runtime error code to a message string
`Profile`	Read or write the registry entries used by Signal
`ProgKill`	Kill an application started with `ProgRun()`
`ProgRun`	Start another application running
`ProgStatus`	Check on a program started using `ProgRun()`
`ScriptRun`	Set the next script to run automatically
`Seconds`	Get or set current relative time in seconds
`Sound`	Play a tone or a .wav file
`Speak`	Convert text into speech
`System`	Get system revision as number
`System$`	Get system name as a string
`Time$`	Get system time in a string in a variety of formats
`TimeDate`	Get system date and time as numbers
`Yield`	Allow the system to catch up on background processing
`YieldSystem`	Surrender current time slice and sleep Signal

5 Alphabetical command reference

This section of the manual lists commands alphabetically. If you are not sure which command you require, look in the *Commands by function* chapter. You might also find the index useful as it cross-references commands and common keywords.

Abs()

This evaluates the absolute value of an expression as a real number. This can also replace the elements of a real or integer array with their absolute values.

```
Func Abs(x|x[]{[]...});
```

`x` A real number, or a real or integer array

Returns If `x` is an array, this returns 0 if all was well, or a negative error code if integer overflow was detected. Otherwise it returns `x` if `x` is positive, otherwise `-x`.

See also: `ATan()`, `Cos()`, `Exp()`, `Frac()`, `Ln()`, `Log()`, `Max()`, `Min()`, `Pow()`, `Rand()`, `Round()`, `Sin()`, `Sqrt()`, `Tan()`, `Trunc()`

App()

Signal is a MDI (multiple document interface) application, and lives in a window. This function returns the application window handle, and some special Signal view handles. Other Signal view handles are returned by functions creating them or finding them.

```
Func App({type%});
```

`type%` This specifies the window handle to return. If omitted the value 0 is used:

-2 The highest view handle currently used by Signal.
-1 100 times the program revision.
0 The application window, so users can resize the application.
1 The Signal system toolbar handle, so it can be hidden and shown.
2 The Signal status bar handle, so it can be hidden and shown.
3 The window handle of the running script (so you can hide it).
4 The Signal edit bar handle, so it can be hidden and shown.
5 The Signal script bar handle, so it can be hidden and shown.
6 The Signal sample configurations bar handle, so it can be hidden and shown.

Returns A handle for the selected window. If the requested window does not exist the return value is 0.

For example:

```
View(App(3));            'make script window the current view
WindowVisible(0);        'hide script window
View(App(0));            'application window is the current view
WindowVisible(3);        'resize the Signal window to mid screen
```

See also: `View()`, `Dup()`, `ViewFind()`, `Window()`, `WindowVisible()`

AppendFrame()

This function appends a new frame to the current data view, which should not be an online sampling view. The new frame will be cleared or can optionally be initialised with a copy of the current frame’s data. The current frame in the view is not changed.

```
Func AppendFrame({copy%});
```

`copy%` If this is present and non-zero the new frame will hold a copy of the current frame’s data.

Returns Zero or a negative error code.

See also: `DeleteFrame()`, `FrameCount()`, `FrameFlag()`, `FrameTag()`

ArrXXX() commands

These functions operate on one dimensional arrays of data, allowing you to use one script step to replace code that would otherwise need a loop such as `repeat...until`, `while...wend` or `for...next`. A loop with the equivalent operations on every item of data takes a lot longer to execute than a Signal array command that does the same thing. You can declare an array variable in your script, or an array can be the items in a channel of a data view which you access using the `View(v,c).[]` construction in place of `fred[]`. The source or destination in the following functions can be data in a view or an array variable declared with the `var` or `proc` or `func` statements.

An array argument can be an array or part of an array as described in detail in the *Arrays of data* section in the *Script language syntax* chapter. The following is a list of the array commands, followed by some examples of how they might appear in a script.

The array commands are:

```
Func ArrAdd(dest[]{[]...},source[]{[]...}|value);
Func ArrConst(dest[]{[]...}, source[]{[]...}|value);
Proc ArrDiff(dest[]);
Func ArrDiv(dest[]{[]...}, source[]{[]...}|value);
Func ArrDivR(dest[]{[]...}, source[]{[]...}|value);
Func ArrDot(source1[],source2[]);
Func ArrFFT(dest[], mode%);
Func ArrFilt(dest[], coef[]);
Func ArrIntgl(dest[]);
Func ArrMul(dest[]{[]...}, source[]{[]...}|value);
Proc ArrSort(sort[]{, opt%{, arr1[]{, arr2[]{, ...}}}});
Proc ArrSpline(dest[], source[]{, ratio{, start}});
Func ArrSub(dest[]{[]...}, source[]{[]...}|value);
Func Func ArrSubR(dest[]{[]...}, source[]{[]...}|value);
Func ArrSum(source[]{, &mean{, &stDev}});
```

`dest` A real or integer array, or a view, that holds the result.
`source` A real or integer array, or a view.
`value` A real or integer value

Integer overflow can be detected with integer destination arrays when the source or value is a real. `ArrAdd()`, `ArrConst()`, `ArrDiv()`, `ArrDivR()`, `ArrFilt()`, `ArrMul()`, `ArrSub()`, `ArrSubR()`, `ArrSum()` can all return a negative error code to indicate integer overflow.

The following are examples of what the function calls can look like in action. These are examples of using all or part of single dimension arrays.

```
var fred[100], tom[100], jim%[200];
var val:=0.5;

ArrAdd(fred[],jim%[]);      'Add elements 0-99 of jim% to fred
ArrSub(fred[],tom[]);       'Subtract each tom from each fred
ArrSubR(fred[]),tom[]);     'The negative of the above result
ArrSub(jim%[],val);         'Subract val from all elements of jim%

ArrAdd(jim%[2:8],10);       'Add value 10 to elements 2-9 of jim%
```

These are examples of using all or part of two dimension arrays.

```
var chans[2][100];          'Array of 2 rows and 100 columns
var data[3][30];            'Array of 3 rows and 30 columns
var jim%[200]

ArrDot(chans[0][],chans[1][]);     'form the dot product of two rows
```

This would set first two elements of column one to the first two elements of `jim%`

```
ArrConst(data[0:2][1],jim%[]);     'copy jim% to one column of data
```

and this would do exactly the same

```
data[0][1]:=jim%[0];
data[1][1]:=jim%[1];
```

These are examples of using array arithmetic functions on data view channel data.

If `vm%` is a data view handle, and `ch%` is a channel number, the following will add the value of the single element `fred[10]` to all elements of channel `ch%` in data view

```
ArrAdd(View(vm%,ch%).[], fred[10]);
```

This will subtract data in channel 2 from channel 1 in data view vm%

```
ArrSub(View(vm%,1).[],View(vm%, 2).[]);
```

See also: `ArrAdd(), ArrConst(), ArrDiff(), ArrDiv(), ArrDivR(), ArrDot(), ArrFFT(), ArrFilt(), ArrIntgl(), ArrMul(), ArrSub(), ArrSubR(), ArrSum(), Len(), View(v,c).[]`

ArrAdd()

This function adds a constant or an array to a real or integer array.

```
Func ArrAdd(dest[]{[]...},source[]{[]...}|value);
```

`dest` The destination array (1 to 5 dimensions).

`source` An array of reals or integers with the same number of dimensions as `dest`. If the dimensions have different sizes, the smaller size is used for each dimension.

`value` A value to be added to all elements of the destination array.

Returns The function returns 0 if all was well, or a negative error code for integer overflow. Overflow is detected when adding a real array to an integer array and the result is set to the nearest valid integer.

In the following examples we assume that the current view is a memory view:

```
var fred[100], jim%[200], two[3][30], add[3][30];
ArrAdd(fred[],1.0);          'Add 1.0 to all elements of fred
ArrAdd(fred[], jim%[]);      'Add elements 0-99 of jim% to fred
ArrAdd(view(0, 1).[],fred[10]); 'Add fred[10] to memory channel 1
ArrAdd(two[][], add[][]);    'Add corresponding elements
```

See also: `ArrXXX(), ArrConst(), ArrDiv(), ArrDivR(), ArrDot(), ArrMul(), ArrSub(), ArrSubR(), ArrSum(), Len(), BuffAdd(), BuffAddTo(), View(v,c).[]`

ArrConst()

This function sets an array or result view to a constant value, or copies the elements of an array or result view to another array or result view. You can copy number or strings. It is an error to attempt to copy numbers to a string array, or strings to a numeric array.

```
Func ArrConst(dest[]{[]...}, source[]{[]...}|value);
```

`dest` The destination array of 1 to 5 dimensions of any type (real, integer, string).

`source` An array with same number of dimensions as `dest`. If the arrays have different sizes, the smaller size is used for each dimension.

`value` A value to be copied to all elements of the destination array.

Returns The function returns 0, or a negative error code. If an integer overflows, the element is set to the nearest integer value to the result.

In the examples below the indices *j* and *i* mean repeat the operation for all values of the indices. `a1d` and `b1d` are vectors, `a2d` and `b2d` are matrices. The arrays and `value` must either both be numeric, or both be strings.

Function	Operation
`ArrConst(a1d[], value);`	`a1d[`*i*`] := value`
`ArrConst(a1d[], b1d[]);`	`a1d[`*i*`] := b1d[`*i*`]`
`ArrConst(a2d[][], value);`	`a2d[`*j*`][`*i*`] := value`
`ArrConst(a2d[][], b2d[][]);`	`a2d[`*j*`][`*i*`] := a2d[`*j*`][`*i*`]`

See also: `ArrXXX()`, `ArrAdd()`, `ArrDiv()`, `ArrDivR()`, `ArrDot()`, `ArrMul()`, `ArrSub()`, `ArrSubR()`, `ArrSum()`, `BuffCopy()`, `BuffCopyTo()`, `BuffExchange()`, `Len()`, `View(v,c).[]`

ArrDiff()

This procedure replaces an array with an array of differences. You can use this as a crude form of differentiation, however `ArrFilt()` provides a better method. See `ArrXXX()` for examples of using arrays as arguments.

```
Proc ArrDiff(dest[]);
```

`dest[]` A real or integer array that is to be replaced by an array of differences. The first element of the array is left unchanged.

The effect of the `ArrDiff()` function can be undone by `ArrIntgl()`. The following two blocks of code perform the same function:

```
var work[100],i%;
...
ArrDiff(work[]);                 'Form differences
...
for i%:=99 to 1 step -1 do       'Form differences the hard way
  work[i%] := work[i%] - work[i%-1];
  next;
```

See also: `ArrXXX()`, `ArrFilt()`, `ArrIntgl()`, `ChanDiff()`, `Len()`, `View(v,c).[]`

ArrDiv()

This function divides a real or integer array by an array or a constant. Use `ArrDivR()` to form the reciprocal of an array. Division by zero and integer overflow are detected.

```
Func ArrDiv(dest[]{[]...}, source[]{[]...}|value)
```

`dest` An array of real or integer values.

`source` An array of reals or integers with the same number of dimensions as `dest`, used as the denominator of the division. If the arrays have different sizes, the smaller size is used for each dimension.

`value` A value used as the denominator of the division.

Returns 0 or a negative error code if integer overflow or division by zero occurs.

If there was integer overflow when assigning the result to an integer array the result is set to the nearest allowed integer value. If division by zero occurs, the associated destination element is not changed.

The function performs the operations listed below. The indices *j* and *i* mean repeat the operation for all values of the indices. Both `a1d` and `b1d` are vectors, `a2d` and `b2d` are matrices. The arrays and `value` may be integer or real.

Function	Operation
`ArrDiv(a1d[], value);`	`a1d[i] := a1d[i] / value`
`ArrDiv(a1d[], b1d[]);`	`a1d[i] := a1d[i] / b1d[i]`
`ArrDiv(a2d[][], value);`	`a2d[j][i] := a2d[j][i] / value`
`ArrDiv(a2d[][], b2d[][]);`	`a2d[j][i] := a2d[j][i] / a2d[j][i]`

See also: `ArrXXX(), ArrAdd(), ArrConst(), ArrDiff(), ArrDivR(), ArrDot(), ArrMul(), ArrSub(), ArrSubR(), ArrSum(), BuffDiv, BuffDivBy(), Len(), View(v,c).[]`

ArrDivR()

This function divides a real or integer array into an array or a constant.

```
Func ArrDivR(dest[]{[]...}, source[]{[]...}|value);
```

`dest` An array of reals or integers used as the denominator of the division and for storage of the result.

`source` A real or integer array with the same number of dimensions as `dest` used as the numerator of the division. If the arrays have different sizes, the smaller size is used for each dimension.

`value` A value used as the numerator of the division.

Returns 0 or a negative error code if integer overflow or division by zero occurs.

If there was integer overflow when assigning the result to an integer array the result is set to the nearest allowed integer value. If division by zero occurs, the associated destination element is not changed.

The function performs the operations listed below. The indices *j* and *i* mean repeat the operation for all values of the indices. Both `a1d` and `b1d` are vectors, `a2d` and `b2d` are matrices. The arrays and `value` may be integer or real.

Function	Operation
`ArrDivR(a1d[], value);`	`a1d[i] := value / a1d[i]`
`ArrDivR(a1d[], b1d[]);`	`a1d[i] := b1d[i] / a1d[i]`
`ArrDivR(a2d[][], value);`	`a2d[j][i] := value / a2d[j][i]`
`ArrDivR(a2d[][], b2d[][]);`	`a2d[j][i] := a2d[j][i] / a2d[j][i]`

See also: `ArrXXX(),ArrAdd(), ArrConst(), ArrDiv(), ArrDot(), ArrMul(), ArrSub(), ArrSubR(), ArrSum(), BuffDiv(), BuffDivBy(), Len(), View(v,c).[]`

ArrDot()

This function multiplies one array by another and returns the sum of the products (sometimes called the dot product of two arrays). The arrays are not changed. See `ArrXXX()` for examples of using arrays as arguments.

```
Func ArrDot(arr1[], arr2[]);
```

`arr1` An array of reals or integers.

`arr2` An array of reals or integers.

Returns The function returns the sum of the products of the corresponding elements of `arr1` and `arr2`.

See also: `ArrXXX(),ArrAdd(), ArrConst(), ArrDiff(), ArrDiv(), ArrDivR(), ArrFFT(), ArrFilt(), ArrIntgl(), ArrMul(), ArrSub(), ArrSubR(), ArrSum(), Len(), View(v,c).[]`

ArrFFT()

This command performs spectral analysis on an array of data. Variants of this command produce log amplitude, linear amplitude, power and relative phase as well as an option to window the original data. See `ArrXXX()` for examples of using arrays as arguments.

```
Func ArrFFT(dest[], mode%);
```

`dest` An array of real numbers to process. The array should be a power of two and at least eight points long. If the number of points is not a power of two, the size is reduced to the next lower power of two points.

`mode%` The mode of the command, in the range 0 to 5. Modes are defined below.

Returns The function returns 0 or a negative error code.

Modes 1 and 3-5 take an array of data that is a set of equally spaced samples in some unit (usually time). If this unit is `xin`, the output is equally spaced in units of 1/`xin`. In the normal case of input equally spaced in seconds, the output is equally spaced in 1/seconds, or Hz. If there are `n` input points, and the interval between the input points is `t`, the spacing between the output points is `1/(n*t)`. The transform assumes that the sampled waveform is composed of sine and cosine waves of frequencies: `0`, `1/(n*t)`, `2/(n*t)`, `3/(n*t)` up to `(n/2)/(n*t)` or `1/(2*t)`.

Display of phase in data views

The phase information sits rather uncomfortably in a data view. When it is drawn, the x axis has the correct increment per bin, but starts at the wrong frequency. If you need to draw it, the simplest solution is to copy the phase information to bin 1 from bin n/2+1 and set bins 0 and n/2 to 0 (this destroys any amplitude information):

```
ArrConst(View(0,1).[1:], View(0,1).[n/2+1:]); 'Copy phase to start
View(0,1).[0]:=0; View(0,1).[n/2]:=0; 'Clear DC and Nyquist points
View(0).Draw(0, MaxTime()/2);                     'Display the phase
```

Mode 0: Window the data

This mode is used to apply a raised cosine window to the data array. See `SetPower()` for an explanation of windows. The selected data is multiplied by a raised cosine window of maximum amplitude 1.0, minimum amplitude 0.0. This window causes a power loss in the result of a factor of 3/8.

You can supply your own window to taper the data, using the array arithmetic commands. The raised cosine is supplied as a general purpose window.

Mode 1: Forward FFT

This mode replaces the data with its forward Fast Fourier Transform. You could use this to remove frequency components, then perform the inverse transform. The output of this mode is in two parts, representing the real and imaginary result of the transform (or the cosine and sine components). The first n/2+1 points of the result hold the amplitudes of the cosine components of the result, the remaining n/2-1 points hold the amplitudes of the sine components. In the case of an 8 point transform, the output has the format:

point	frequency	contents	point	frequency	contents
0	DC(0)	amplitude	4	4/(n*t)	cosine amplitude
1	1/(n*t)	cosine amplitude	5	1/(n*t)	sine amplitude
2	2/(n*t)	cosine amplitude	6	2/(n*t)	sine amplitude
3	3/(n*t)	cosine amplitude	7	3/(n*t)	sine amplitude

There is no sine amplitude at a frequency of 4/(n*t) as this sine wave has amplitude 0 at all sampled points.

Mode 2: Inverse FFT

This mode takes data in the format produced by the forward transform and converts it back into a time series. In theory, the result of mode 1 followed by mode 2, or mode 2 followed by mode 1, would be the original data. However, each transform adds some noise due to rounding effects in the arithmetic, so the transforms do not invert exactly.

One use of modes 1 and 2 is to filter data. For example, to remove high frequency noise use mode 1, set unwanted frequency bins to 0, and use mode 2 to reconstruct the data.

Mode 3: dB and phase

This mode produces an output with the first n/2+1 points holding the log amplitude of the power spectrum in dB, and the second n/2-1 points holding the phase (in radians) of the data. In the case of our 8 point transform the output format would be:

point	frequency	contents	point	frequency	contents
0	DC	log amplitude in dB	4	4/(n*t)	log amplitude in dB
1	1/(n*t)	log amplitude	5	1/(n*t)	phase in radians
2	2/(n*t)	log amplitude	6	2/(n*t)	phase in radians
3	3/(n*t)	log amplitude	7	3/(n*t)	phase in radians

There is no phase information for DC or for the point at 4/(n*t) because the phase for both of these is zero. If you want phase in degrees, multiply by 57.3968 . The log amplitude is calculated by taking the result of a forward FFT (same as mode 1 above) and forming:

$dB = 10.0\ \mathrm{Log}(power)$ Power as defined for mode 5

Mode 4: Amplitude and phase

This mode produces the same output format as mode 3, but with amplitude in place of log amplitude. The amplitude is calculated by taking the result of a forward FFT (same as mode 1 above), and forming:

$amplitude = (\cos^2+\sin^2)^{0.5}$

Mode 5: Power and phase

This mode produces the same output format as modes 2 and 3, but with the result in terms of RMS power. The power is calculated by taking the result of a forward FFT (same as mode 1 above), and forming:

$power = (\cos^2+\sin^2) * 0.5$ for all components except the DC and Nyquist
$power = \mathrm{DC}^2$ or $\mathrm{Nyquist}^2$ for the DC and Nyquist components

You can compare the output of this mode with the result of `SetPower()`. If you have a waveform channel on channel 2 in view 1, with at least 1024 data points, do the following:

```
var dView%,spView%, m%;          'assume we are in a file view
var ch%:=2;                      'use data from channel 2
var xRes;                        'x resolution to apply to result

dView%:=View(0);                 'store handle for data view
spView% := SetPower(ch%,1024);   'select power spectrum channel 1
Process(0);                      'process first 1024 data points
WindowVisible(1);                'make new memory view visible
xRes:=BinSize(1);                'get spectrum resolution in Hz
m%:=SetMemory(1,1024,xRes,0,0,0,0,"FFT","Hz","Volt^2","","Power");
' created memory view ready to hold 1024 points for transformation
WindowVisible(1);                'make new memory view visible
ArrConst(View(m%,1).[],View(dView%,ch%).[]);   'copy channel data
ArrFFT(View(m%,1).[], 0);        'apply raised cosine window to it
ArrFFT(View(m%,1).[], 5);                      'take power
spectrum
ArrMul(View(m%,1).[0:513], 4.0/3.0);           'adjust amplitude
Draw(0,500*BinSize(1));  Optimise(1);   'show 500 bins of power
View(spView%);                          'look at SetPower() result
Draw(0,500*BinSize(1)); Optimise(1);    'show same bins of power
```

The two results are identical. The view generated by `ArrFFT()` would be 3/8 of the amplitude of the view generated by `SetPower()`. The reason for the difference is that the `SetPower()` command compensates for the effect of the window it uses internally by multiplying the result by 8/3. To produce the same numeric result, multiply by 8/3.

Note: The behaviour of the ArrFFT function in mode 5 (Power and Phase) was altered in version 1.60 of Signal. Previous versions produced a result which was 3/4 of the `SetPower()` result.

See also: `ArrXXX()`, `ArrAdd()`, `ArrConst()`, `ArrDiff()`, `ArrDiv()`, `ArrDivR()`, `ArrDot()`, `ArrFilt()`, `ArrIntgl()`, `ArrMul()`, `ArrSub()`, `ArrSubR()`, `ArrSum()`, `Len()`, `SetPower()`, `View(v,c).[]`

ArrFilt()

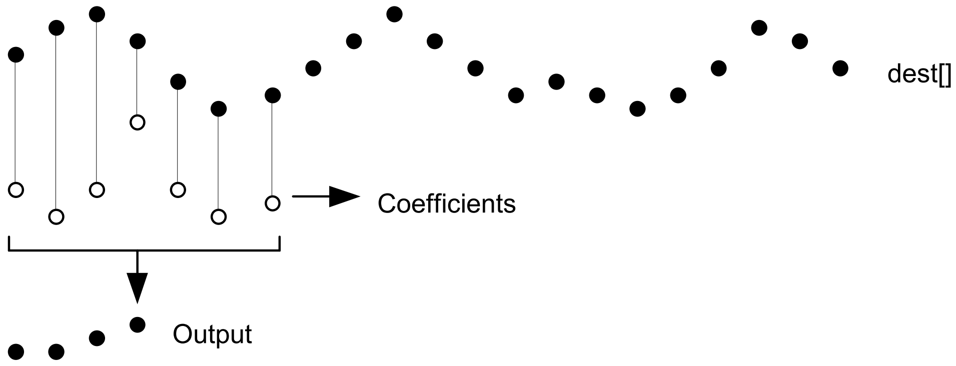
This function filters a real array by replacing each element by the product of a coefficient array and the surrounding elements of the original array. This implements a FIR (Finite Impulse Response) filter. See `ArrXXX()` for examples of using arrays as arguments.

```
Func ArrFilt(dest[], coef[]);
```

`dest[]` A real array holding the data to be filtered. It is replaced by the filtered data.

`coef[]` A real array of filter coefficients. This array is usually an odd number of data points long so that the result is not phase shifted.

Returns The function returns 0 if there was no error, or a negative error code.

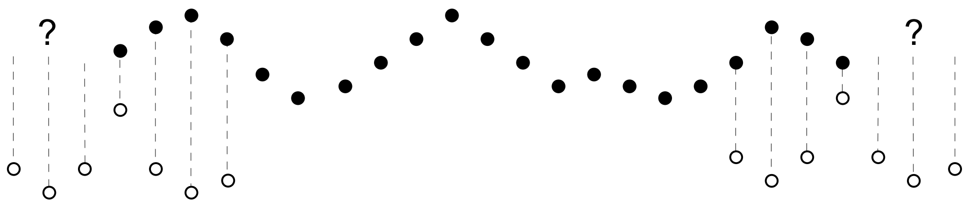


This diagram shows the general principle of the FIR filter. The hollow circles represent the filter coefficients, and the solid circles are the input and output waveforms. Each output point is generated by multiplying the waveform by the coefficients and summing the result. The coefficients are then moved one step to the right and the process repeats.

From this description, you can see that the filter coefficients (from right to left) are the *impulse response* of the filter. The impulse response is the output of a filter when the input signal is all zero except for one sample of unit amplitude.

In the example above with seven coefficients, there is no time shift caused by the filter. If the filter uses an even number of coefficients, there is a time shift in the output of half a sample period.

The filter operation is applied to every element of the array. To do this at the start and end of the array there is a problem as some of the coefficients have no corresponding data element.



The simple solution is to take these missing points as copies of the first and last points. This is usually better that taking these points as 0. You should remember that the first and last (*nc*+1)/2 points are unreliable, where *nc* is the number of coefficients.

A simple use of this command is to produce three point smoothing of data, replacing each point by the mean of itself and the two points on either side:

```
var data[1000],coef[3];    'arrays of data and the coefficients
...                        'fill data[] with values
ArrConst(coef[],0.33333);  'set all three coefficients to 0.33333
ArrFilt(data[],coef[]);    'smooth the data.
```

A more complicated example would be to implement a differentiator to calculate the slope or gradient of an array. The simplest case is to use two points:

```
coef[0]:=-1; coef[1]:=1;      'simple difference
ArrFilt(data[], coef[0:1]);  'for differences, equivalent to...
ArrDiff(data[]);              '... just using the differences
```

A simple difference produces a very crude differentiator. A slightly better one, with three coefficients is:

```
coef[0] := -0.5; coef[1] := 0.0; coef[2] := 0.5;
ArrFilt(data[], coef[]);
```

You can improve the result with more points, for example for 4 points, the coefficients are -0.3, -0.1, 0.1, 0.3 and for five points try -0.2, -0.1, 0.0, 0.1, 0.2. It is more usual to use an odd number of points as this does not cause a shift of the result by half a point.

See also: `ArrXXX(),ArrAdd(), ArrConst(), ArrDiv(), ArrDivR(), ArrDot(), ArrFFT(), ArrMul(), ArrSub(), ArrSubR(), ArrSum(), ChanSmooth(), Len(), View(v,c).[]`

ArrIntgl()

This procedure is the inverse of `ArrDiff()`, replacing each point by the sum of the points from the start of the array up to the element. See `ArrXXX()` for examples of using arrays as arguments.

```
Proc ArrIntgl(dest[]);
```

`dest` An array or real or integer data.

Each element of the array is replaced by the sum of all the elements up to and including that element. The function is equivalent to the following:

```
for i%:=1 to Len(dest[])-1 do
    dest[i%] := dest[i%] + dest[i%-1];
```

See also: `ArrXXX(),ArrDiff(), ChanIntgl(), Len(), View(v,c).[]`

ArrMul()

This command is used to form the product of a pair of arrays, or to scale an array by a constant. A less obvious use is to negate an array by multiplying by -1.

```
Func ArrMul(dest[]{[]...}, source[]{[]...}|value);
```

`dest` An array of real or integer numbers. If `dest` is integer, the multiplication is done as reals and truncated to integer.

`source` A real or integer array with the same number of dimensions as `dest`. If the arrays have different sizes, the smaller size is used for each dimension.

`value` A value to multiply the data in `dest`.

Returns The function returns 0 if all was well, or a negative error code.

The function performs the operations listed below. The indices *j* and *i* mean repeat the operation for all values of the indices. Both `a1d` and `b1d` are vectors, `a2d` and `b2d` are matrices. The arrays and `value` may be integer or real.

Function	Operation
`ArrMul(a1d[], value);`	`a1d[i] := a1d[i] * value`
`ArrMul(a1d[],b1d[]);`	`a1d[i] := a1d[i] * b1d[i]`
`ArrMul(a2d[][],value);`	`a2d[j][i] := a2d[j][i] * value`
`ArrMul(a2d[][],b2d[][]);`	`a2d[j][i] := a2d[j][i] * a2d[j][i]`

See also: `ArrXXX(),ArrAdd(), ArrConst(), ArrDiv(), ArrDivR(), ArrDot(), ArrSub(), ArrSubR(), ArrSum(), BuffMul(), BuffMulBy(), Len(), View(v,c).[]`

ArrSort()

This function sorts an array of any data type and optionally orders additional arrays to match the sorted array.

```
Proc ArrSort(sort[]{, opt%{, arr1[]{, arr2[]{, ...}}}});
```

`sort[]` An array of integer, real or string data to sort.

`opt%` This optional argument holds the sorting options. If omitted, the value 0 is used. It is the sum of the following flag values:

1 Sort in descending order. If omitted, the data is sorted in ascending order.

2 Case sensitive sort when `sort[]` is an array of strings. String sorts are usually case insensitive. If omitted, the sort is case insensitive.

`arrn[]` `If present, these arrays are sorted in the same order as the sort[] array. The arrays can be of any type. You can sort up to 18 additional arrays.`

See also: `ArrXXX(), ArrAdd(), ArrConst(), ArrDiff(), ArrDiv(), ArrDivR(), ArrDot(), ArrMul(), ArrSubR(), ArrSum(), BuffSub(), BuffSubFrom(), Len(), View(v,c).[]`

ArrSpline()

This function interpolates an array of real or integer data sampled at one rate into another array sampled at a different rate using cubic splines. This assumes that the source data has continuous first and second derivatives and that the second derivatives vary linearly from point to point. The second derivatives at the first and last point of the source data are set to zero. We interpolate from up to one input sampling interval before the first source point to up to one sampling interval after the last source point.

```
Proc ArrSpline(dest[], source[]{, ratio{, start}});
```

`dest` A real or integer vector that holds the interpolated result.

`source` A vector that is integer if `dest` is integer or real if `dest` is real.

`ratio` If present, this is the ratio of the sampling interval of `dest` to the sampling interval of `source` (or the `source` frequency divided by the `dest` frequency). If omitted, we assume that the first and last points of the two arrays are aligned. There are no restrictions on `ratio`, it can even be negative (in which case the output is backwards relative to the input).

`start` If present, this is the position of the first point of `dest` relative to the first point of `source` in units of the sample interval of the source array. If omitted, it is assumed to be 0 (the first points of both arrays fall at the same position).

You will get the best results if you can supply source data before and after the output time range. The effect of a source point on the interpolation of an interval `n` points away falls by a factor of approximately 4 each time `n` increases by 1. There is rarely the need to supply more than 15 data points before and after the interpolation range.

An example

Suppose we have a source vector of 100 points sampled at 100 Hz, with the first point sampled at 5.02 seconds, and we want to generate an array sampled at 1000 Hz that starts at 5.335 and lasts 0.5 seconds. In this case, the value of `ratio` is 0.001/0.01, which is 0.1. The value of `start` is 31.5, which is the time difference (5.335 - 5.02) divided by 0.01, the sample interval of the source channel.

See also: `DrawMode()`

ArrSub()

This function forms the difference of two arrays or subtracts a constant from an array. If the destination is an integer array, overflow is detected when subtracting real values.

```
Func ArrSub(dest[]{[]...}, source[]{[]...}|value);
```

`dest` A real or integer array that holds the result.

`source` A real or integer array with the same number of dimensions as `dest`. If the arrays have different sizes, the smaller size is used for each dimension.

`value` A real or integer value.

Returns 0 if all is well or a negative error code if integer overflow is detected.

The function performs the operations listed below. The indices *j* and *i* mean repeat the operation for all values of the indices. Both `a1d` and `b1d` are vectors; `a2d` and `b2d` are matrices. The arrays and `value` may be integer or real.

Function	Operation
`ArrSub(a1d[], value);`	`a1d[i] := a1d[i] - value`
`ArrSub(a1d[],b1d[]);`	`a1d[i] := a1d[i] - b1d[i]`
`ArrSub(a2d[][],value);`	`a2d[j][i] := a2d[j][i] - value`
`ArrSub(a2d[][],b2d[][]);`	`a2d[j][i] := a2d[j][i] - a2d[j][i]`

See also: `ArrXXX(),ArrAdd(), ArrConst(), ArrDiff(), ArrDiv(), ArrDivR(), ArrDot(), ArrMul(), ArrSubR(), ArrSum(), BuffSub(), BuffSubFrom(), Len(), View(v,c).[]`

ArrSubR()

This function forms the difference of two arrays or subtracts an array from a constant. If the destination is an integer array, overflow is detected when subtracting real values.

```
Func ArrSubR(dest[]{[]...}, source[]{[]...}|value);
```

`dest` A real or integer array.

`source` A real or integer array with the same number of dimensions as `dest`. If the arrays have different sizes, the smaller size is used for each dimension.

`value` A real or integer value.

Returns 0 if all is well or a negative error code if integer overflow is detected.

The function performs the operations listed below. The indices *j* and *i* mean repeat the operation for all values of the indices. Both `a1d` and `b1d` are vectors, `a2d` and `b2d` are matrices. The arrays and `value` may be integer or real.

Function	Operation
`ArrSubR(a1d[], value);`	`a1d[i] := value - a1d[i]`
`ArrSubR(a1d[],b1d[]);`	`a1d[i] := b1d[i] - a1d[i]`
`ArrSubR(a2d[][],value);`	`a2d[j][i] := value - a2d[j][i]`
`ArrSubR(a2d[][],b2d[][]);`	`a2d[j][i] := a2d[j][i] - a2d[j][i]`

See also: `ArrXXX(),ArrSub(), ArrSum(), BuffSub(), BuffSubFrom(), Len(), View(v,c).[]`

ArrSum()

This function forms the sum of the values in an array, and optionally forms the mean and standard deviation. See `ArrXXX()` for examples of using arrays as arguments.

```
Func ArrSum(arr[]|arr%[]{, &mean{, &stDev}});
```

`arr` A real or integer array to process.

`mean` If present it is returned holding the mean of the values in the array. The mean is the sum of the values divided by the number of array elements.

`stDev` If present, this is returns the standard deviation of the array elements. If the array has only one element the result is 0. It is equivalent to:

```
Func StDev(arr[])
var n%, i%, mean, sumSq:=0.0, temp;
n% := Len(arr[]);        'get array size
mean := ArrSum(arr[])/n%;'mean value
for i%:=0 to n%-1 do     'form sum of squares
    temp := arr[i%] - mean;
    sumSq := sumSq + temp * temp;
    next;
if (n%>1) then
    return Sqrt(sumSq/(n%-1));
else
    return 0.0;
endif;
end;
```

Returns The function returns the sum of the array elements.

See also: `ArrXXX(),ArrAdd(), ArrConst(), ArrDiff(), ArrDiv(), ArrDivR(), ArrDot(), ArrFFT(), ArrFilt(), ArrIntgl(), ArrMul(), ArrSub(), ArrSubR(), Len(), View(v,c).[]`

Asc()

This function returns the ASCII code of the first character in the string as an integer.

```
Func Asc(text$);
```

`text$` The string to process.

See also: `Chr$(), DelStr$(), LCase$(), Left$(), Len(), Mid$(), Print$(), Right$(), Str$(), UCase$(), Val()`

ATan()

This function returns the arc tangent of an expression, or the arc tangent of an array:

```
Func ATan(s|s[]{[]...} {,c});
```

`s` If the only argument, the function uses this for the arc tangent calculation. `s` can also be a real array (in which case `c` must not be present).

`c` If this is present, the function uses `s/c` for the calculation.

Returns If `s` is an array, each element of `s` is replaced by its arc tangent in the range $-\pi/2$ to $\pi/2$ radians. The function returns 0 if all was well or a negative error code.

When `s` is not an array, if `s` is the only argument, the function returns the arc tangent of `s` in the range $-\pi/2$ to $\pi/2$. If `c` is present, the function calculates the result of `ATan(s/c)` and uses the signs of `s` and `c` to decide the quadrant of the result. With the second argument, the result is in the range $-\pi$ to π.

Arc sine of a single value: `s` can be calculated as: `ATan(s/Sqrt(1-s*s))`.

Arc cosine can be calculated as: `ATan(Sqrt(1-s*s)/s)`.

See also: `Cos(), Cosh(), Ln(), Log(), Pow(), Sin(), Sinh(), Sqrt(),Tan()`

BinError()

This function is used in a memory or file view, with error bins enabled, to access the error information. Error bins are created for a memory view created with `SetAverage()` or `SetAutoAv()` with the last argument set to 1 and are subsequently stored in the cfs file if the memory view is saved. If you are setting the error information you must set the sweep count with `Sweeps()` first, as the sweep count is used to convert the standard deviation into the internal storage format. There are two command variants: the first transfers data for a single bin, the second for an array of bins:

```
Func BinError(chan%, bin% {,newSD});
Func BinError(chan%, bin%, sd[]{, set%});
```

`chan%` The channel number in the file or memory view.

`bin%` The first bin number for which to get or set the error information.

`newSD` If present, this sets the standard deviation for a single bin.

`sd[]` An array used to hold standard deviation values. Values are transferred starting at bin `bin%` in the file or memory view. If the array is too long, extra bins are ignored.

`set%` If present and non-zero, the values in `sd[]` are copied to the memory view. If omitted or zero, values are copied from the file or memory view into `sd[]`.

Returns The first command variant returns the standard deviation at the time of the call. The second variant returns the number of bins copied. If there are 0 or 1 sweeps of data or errors are not enabled, the result is 0.

To illustrate how errors are calculated, we will assume that we are dealing with an average that is set to display the mean of the data in each bin. In terms of the script language, if the array `s[]` holds the contribution of each sweep to a particular bin, the mean, standard deviation and standard error of the mean are calculated as follows:

```
var mean, sd:=0, i%, diff, sem;
for i%:= 0 to Sweeps()-1 do
    mean += s[i%];              'form sum of data
    next;
mean /= Sweeps();               'form mean data value
for i%:= 0 to Sweeps()-1 do
    diff := s[i%]-mean;         'difference from mean
    sd += diff*diff;            'sum squares of differences
    next;
sd := Sqrt(sd/(Sweeps()-1));    'the standard deviation
sem := sd/Sqrt(Sweeps());       'the standard error of the mean
```

We divide by `Sweeps()-1` to form the standard deviation because we have lost one degree of freedom through calculating the mean from the data.

See also: `BinSize(), BinToX(), SetAutoAv(), SetAverage(), Sweeps(), XToBin()`

BinSize()

The value returned by this function is normally the x axis increment per point but depends upon the channel type. You can set the bin size in a memory view only.

```
Func BinSize(chan%, {nSize});
```

`chan%` The channel number (1 to n) for which to return information.

`nSize` If this is present it sets a new x axis resolution in a memory view.

Returns The returned value is negative if the channel doesn't exist. Otherwise the value returned depends on the channel type:

Waveform	This is the x axis interval between points on the channel. For sampled data this is the sample interval.
Marker	The underlying x axis resolution of the channel.

See also: `BinToX()`, `XToBin()`, `SampleRate()`

BinToX()

This function converts between bin numbers and x axis units in the current view.

```
Func BinToX(chan%, bin);
```

`chan%` The channel (1 to n) for which to return information.

`bin` A bin number in the view. You can give a non-integer bin number without error. If you give a bin number outside the range of the view, the bin number is limited to the range of the view.

Returns The returned value is zero if the channel doesn't exist. Otherwise it is the equivalent x axis position.

See also: `BinSize()`, `XToBin()`, `BinZero()`

BinZero()

This function returns the x axis position for the first bin in the frame on the given channel. In a memory view you can also set this.

```
Func BinZero(chan%{,offset});
```

`chan%` The channel (1 to n) for which to return information.

`offset` If this is provided it sets a new x axis position for the first data point for channels in a memory view.

Returns The returned value is zero if the channel doesn't exist. Otherwise it is the equivalent x axis position of the start of the data.

See also: `BinSize()`, `BinToX()`, `XToBin()`

BRead()

This reads data into variables and arrays from a binary file opened by `FileOpen()`. The function reads 32-bit integers, 64-bit IEEE real numbers and zero-terminated strings.

```
Func BRead(&arg1|arg1[]|&arg1%|arg1%[]|&arg1$|arg1$[] {,...});
```

`arg` Up to 20 arguments of any type. Signal reads a block of memory equal in size to the combined size of the arguments and copies it into the arguments. Strings or string arrays are read a byte at a time until a zero byte is read.

Returns It returns the number of data items for which complete data was read. This will be less than the number of items in the list if end of file was reached. If an error occurred during the read, a negative code is returned.

See also: `FileOpen(), BReadSize(), BRWEndian(), BSeek(), BWrite()`

BReadSize()

This converts data into variables and arrays from a binary file opened by `FileOpen()`. The function reads 8, 16 and 32-bit integers and converts them to 32-bit integers, and 32 and 64-bit IEEE real numbers and converts them to 64-bit reals. It also reads strings from fixed-size regions in the file (zeros are ignored during the read). The read is from the current file position. The current position after the read is the byte after the last byte read.

```
Func BReadSize(size%, &arg|arg[]|&arg%|arg%[]|&arg$|arg$[]{,...});
```

`size%` The bytes to read for each argument. Legal values depend on the argument type:

Integer	1, 2 or 4	Read 1, 2 or 4 bytes and sign extend to 32-bit integer.
	-1, -2	Read 1 or 2 bytes and zero extend to 32-bit integer.
Real	4	Read 4 bytes as 32-bit real, convert to 64-bit real.
	8	Read 8 bytes as 64-bit real.
String	n	Read n bytes into a string. Null characters end the string.

`arg` Up to 19 target variable(s) to be filled with data. `size%` applies to all targets.

Returns It returns the number of data items for which complete data was read. This will be less than the number of items in the list if end of file was reached. If an error occurred during the read, a negative code is returned.

See also: `FileOpen(), BRead(), BRWEndian(), BSeek(), BWrite()`

BRWEndian()

This gets and sets the "endianism" of binary data files. This affects numeric data used with `BRead()`, `BReadSize()`, `BWrite()` and `BWriteSize()`. PC programs normally use little-endian data (least significant byte at lowest address). Some systems, including the Macintosh, use big-endian data (most significant byte at lowest address). Binary files are little-endian by default.

Most users do not need to use this routine. Only use it if you are writing binary files for use on a big-endian computer or reading binary files that were generated with a big-endian system.

```
Func BRWEndian({new%});
```

`new%` Omit or set –1 for no change. Set 0 for little-endian and 1 for big-endian.

Returns The current endianism as 0 for little, 1 for big or a negative error code.

See also: `FileOpen(), BRead(), BReadSize(), BSeek(), BWrite(), BWriteSize()`

BSeek()

This function moves and reports the current position in a binary file opened by `FileOpen()`. The next binary read or write operation to the file starts from the position returned by this function.

```
Func BSeek({pos% {, rel%}});
```

`pos%` The new file position. Positions are measured in terms of the byte offset in the file from the start, the current position, or from the end. If a new position is not given, the position is not changed and the function returns the current position.

`rel%` This determines to what the new position is relative.
0 Relative to the start of the file (same as omitting the argument).
1 Relative to the current position in the file.
2 Relative to the end of the file.

Returns The new file position relative to the start of the file or a negative error code.

See also: `FileOpen()`, `BReadSize()`, `BRead()`, `BWrite()`

BuffXXX() Buffer commands

The `Buff…` family of commands can be used to carry out arithmetic on sets of data frames using the built-in frame buffer. The frame buffer is an extra frame of data attached to a data document that is provided automatically by Signal. This can be used to hold the results of arithmetic on frames, or to modify the document data. To help to avoid confusion, commands that modify buffer data all have a simple name such as `BuffSub`, or `BuffCopy`, while commands that modify the document frame data have qualified names such as `BuffSubFrom` or `BuffCopyTo`.

Nearly all of the `BuffXXX` commands require a `frame%` argument. This specifies the frame in the data document that is to be used, if omitted, the current frame is used. The current frame in the view is not changed.

The buffer commands do not have channel specification arguments as they operate on all channels. If you require more precise control over frame arithmetic operations, this can be achieved by creating an invisible view to act as a buffer using `SetCopy()` or `SetMemory()`, and then manipulating the frame data directly.

You can access the built-in interactive support for using the frame buffer from the analysis menu or by using the multiple frame dialog.

See also: `ShowBuffer(), BuffAdd(), BuffAddTo(), BuffAcc(), BuffClear(), BuffCopy(), BuffCopyTo(), BuffDiv(), BuffDivBy(), BuffExchange(), BuffMul(), BuffMulBy(), BuffSub(), BuffSubFrom(), BuffUnAcc(), SetCopy(), SetMemory()`

BuffAdd()

This adds the specified frame data to the frame buffer for the current view document.

```
Func BuffAdd({frame%});
```

`frame%` The frame number in the data document for the current view. If omitted, the current frame in the view is used.

Returns Zero or a negative error code.

See also: `ArrAdd(), View(v,c).[], BuffXXX()`

BuffAddTo()

This adds the frame buffer for the current view document to the specified frame data.

```
Func BuffAddTo({frame%});
```

`frame%` The frame number in the data document for the current view. If omitted, the current frame in the view is used.

Returns Zero or a negative error code.

See also: `ArrAdd(), View(v,c).[], BuffXXX()`

BuffAcc()

This adds the specified frame data to an average in the frame buffer for the current view document. The addition is carried out in such a way as to maintain the data as an average, which can be e.g. subtracted from data frames. If you mix `BuffAcc()` and `BuffAdd()` operations, the overall effect will probably be rather messy.

```
Func BuffAcc({frame%});
```

`frame%` The frame number in the data document for the current view. If omitted, the current frame in the view is used.

Returns Zero or a negative error code.

See also: `ArrAdd(), BuffUnAcc(), View(v,c).[], BuffXXX()`

BuffClear()

This clears the data in the frame buffer for the current view document.

```
Func BuffClear();
```

Returns Zero or a negative error code.

See also: `ArrConst(), View(v,c).[], BuffXXX()`

BuffCopy()

This copies the specified frame data to the frame buffer for the current view document.

```
Func BuffCopy({frame%});
```

`frame%` The frame number in the data document for the current view. If omitted, the current frame in the view is used.

Returns Zero or a negative error code.

See also: `ArrConst(), View(v,c).[], BuffXXX()`

BuffCopyTo()

This copies the frame buffer data for the current view document into the specified data frame.

```
Func BuffCopyTo({frame%});
```

`frame%` The frame number in the data document for the current view. If omitted, the current frame in the view is used.

Returns Zero or a negative error code.

See also: `ArrConst(), View(v,c).[], BuffXXX()`

BuffDiv()

This divides the frame buffer for the current view document by the specified frame data.

```
Func BuffDiv({frame%});
```

`frame%` The frame number in the data document for the current view. If omitted, the current frame in the view is used.

Returns Zero or a negative error code.

See also: `ArrDiv(), View(v,c).[], BuffXXX()`

BuffDivBy()

This divides the specified frame data by the frame buffer for the current view document.

```
Func BuffDivBy({frame%});
```

`frame%` The frame number in the data document for the current view. If omitted, the current frame in the view is used.

Returns Zero or a negative error code.

See also: `ArrDiv(), View(v,c).[], BuffXXX()`

BuffExchange()

This exchanges the specified frame data with the frame buffer data for the current view document.

```
Func BuffExchange({frame%});
```

`frame%` The frame number in the data document for the current view. If omitted, the current frame in the view is used.

Returns Zero or a negative error code.

`See also: ArrConst(), View(v,c).[], BuffXXX()`

BuffMul()

This multiplies the frame buffer for the current view document by the data in the specified frame.

```
Func BuffMul({frame%});
```

`frame%` The frame number in the data document for the current view. If omitted, the current frame in the view is used.

Returns Zero or a negative error code.

See also: `ArrMul(), View(v,c).[], BuffXXX()`

BuffMulBy()

This multiplies the specified frame data by the frame buffer for the current view document.

```
Func BuffMulBy({frame%});
```

`frame%` The frame number in the data document for the current view. If omitted, the current frame in the view is used.

Returns Zero or a negative error code.

`See also: ArrMul(), View(v,c).[], BuffXXX()`

BuffSub()

This subtracts the specified frame data from the frame buffer for the current view document.

```
Func BuffSub({frame%});
```

`frame%` The frame number in the data document for the current view. If omitted, the current frame in the view is used.

Returns Zero or a negative error code.

See also: `ArrSub(), View(v,c).[], BuffXXX()`

BuffSubFrom()

This subtracts the frame buffer for the current view document from the specified frame data.

```
Func BuffSubFrom({frame%});
```

`frame%` The frame number in the data document for the current view. If omitted, the current frame in the view is used.

Returns Zero or a negative error code.

See also: `ArrSub(), View(v,c).[], BuffXXX()`

BuffUnAcc()

This subtracts the specified frame data from an average in the frame buffer for the current view document. The arithmetic is carried out in such a way as to maintain the data as an average with the specified frame now not included. If the frame was never included in the average, or if you mix `BuffUnAcc()` and `BuffSub()` operations, the overall effect will probably be rather messy.

```
Func BuffUnAcc({frame%});
```

`frame%` The frame number in the data document for the current view. If omitted, the current frame in the view is used.

Returns Zero or a negative error code.

See also: `ArrSub(), BuffAcc(), View(v,c).[], BuffXXX()`

BWrite()

This function writes binary data values and arrays into a binary file opened or created by `FileOpen()`. The function can write 32-bit integers, 64-bit IEEE real numbers and strings. The output is at the current position in the file. The current position after the write is the byte after the last byte written.

```
Func BWrite(arg1 {,arg2 {,...}});
```

`arg` Up to 20 arguments of any type, including arrays. Signal fills a block of memory equal in size to the combined size of the arguments with the data held in the arguments and copies it to the file.

An integer uses 4 bytes and a real uses 8 bytes. A string is written as the bytes in the string and plus an extra zero byte to mark the end. Use `BWriteSize()` to write a fixed number of bytes.

Returns It returns the number of arguments for which complete data was written. If an error occurred during the write, a negative code is returned.

See also: `FileOpen(), BRead(), BReadSize(), BRWEndian(), BWriteSize()`

BWriteSize()

This writes variables or arrays as binary data into a file opened or created by `FileOpen()`. The function can write 8, 16 and 32-bit integers, 32 and 64-bit reals and strings. This allows you to write formats other than the 32-bit integer and 64-bit real used internally by Signal and to write variable-length strings into fixed size fields in a binary file.

```
Func BWriteSize(size%, arg1 {,arg2 {,...}});
```

`size%` Bytes to write for each argument (or array element if the argument is an array). Legal values depend on the argument type:

Integer	1, 2	Write least significant 1 or 2 bytes.
	4	Write all 4 bytes of the integer.
Real	4	Convert to 32-bit real and write 4 bytes.
	8	Write 8 bytes as 64-bit real.
String	n	Write n bytes. Pad with zeros if the string is too short.

`arg` Up to 19 target variable(s) to be filled with data. `size%` applies to all targets.

Returns It returns the number of data items for which complete data was written or a negative error code.

See also: `FileOpen(), BRead(), BReadSize(), BRWEndian(), BWrite()`

Ceil()

Returns the next higher integral number of the real number or array. `Ceil(4.7)` is `5.0`, `Ceil(4)` is `4`. `Ceil(-4.7)` is `-4`.

```
Func Ceil(x|x[] {[]…});
```

`x` A real number or a real array.

Returns When the argument is an array, the function replaces the array with the next higher integral number of all the points and returns either a negative error code or 0 if all was well.
When the argument is not an array the next higher integral number.

See also: `Abs(),ATan(),Cos(),Exp(),Floor(),Frac(),Ln(),Log(),Max(), Min(),Pow(),Rand(),Round(),Sin(),Sqrt(),Tan(),Trunc()`

Chan$()

This function converts a channel number or a list of channel numbers into a string. Future versions of Signal may have channels types that appear not just as a number on the screen. If a channel does not exist in the current view, it is represented as a number.

```
Func Chan$(chan%|chan%[]);
```

`chan%` Either a channel number or an array of integers in the same format as a channel specification (the first element holds the number of items, followed by the channel numbers).

Returns A channel specification string, for example `"1,3,5..8"`.

See also:`ChanList()`

ChanAdd()

This will add the data from one channel to one or more other channels.

```
Func ChanAdd(cSpc, src%);
```

`cSpc` A channel list specifier of the channels to add data to. See the *Script language syntax* chapter for a definition of channel specifiers.

`src%` The number of the channel containing the data to add.

Returns Zero.

See also: `ChanSub()`, `ChanMult()`, `ChanDiv()`

ChanColour()

This returns and optionally sets the colour of a channel in a file or memory view. This colour overrides the application colour set for the drawing mode of the channel.

```
Func ChanColour(chan%, item%{, col%});
```

`chan%` A channel in the file or memory view.

`item%` The colour item to get and optionally set; 0=background, 1=primary, 2=secondary colour.

`col%` If present, the new colour index for the item. There are 40 colours in the palette, indexed 0 to 39. Use -1 to revert to the application colour for the drawing mode.

Returns The palette colour index at the time of the call, –1 if no colour is set or a negative error code if the channel does not exist.

See also: `Colour()`, `PaletteGet()`, `PaletteSet()`, `XYColour()`

ChanCount()

This counts channels in a data or XY view.

```
Func ChanCount({chan%});
```

`chan%` If present, this specifies the channels to count (this is ignored for XY views). If omitted, the total channel count is returned. It can be -1 for all channels, -2 for all visible channels, -3 for all selected channels, -4 for waveform channels or -5 for marker channels, -6 for selected waveform channels or visible if none selected, -7 for visible waveform channels, -8 for selected waveform channels or –9 for idealised traces.

Returns The returned value is the number of channels of the specified type.

See also: `ChanList()`

ChanDelete()

This function deletes a channel from an XY view or an idealised trace from a file or memory view. You have the option of having the user confirm the deletion. You cannot delete the last XY channel as XY views must always have at least one channel. Channels in XY views are always numbered consecutively, so if you delete a channel, the channel numbers of any higher numbered channels will change. Changes to the XY data will not become permanent until the XY view is saved. It is not possible however, to recover data deleted from an idealised trace channel.

```
Func ChanDelete(chan% {,query%});
```

`chan%` The channel to delete.

`query%` If present and non-zero, the user is asked to confirm the channel deletion if the channel is part of a saved XY data file or an idealised trace channel in a file of memory view.

Returns 0 if the channel was deleted or a negative error code if the user cancelled the operation or tried to delete the last XY channel or for other problems.

See also: `XYDelete(), XYSetChan()`

ChanDiff()

This differentiates the data in specified waveform channels of the current frame in the current view. If the specified channel is not a waveform then the command has no effect. If the frame buffer is being shown, the frame buffer data is used instead. All data points in the channels used are modified.

```
Func ChanDiff(cSpc);
```

`cSpc` A channel list specifier of the channels to differentiate. See the *Script language syntax* chapter for a definition of channel specifiers.

Returns 0 or a negative error code.

See also: `ShowBuffer(), ChanShow(), ChanZero(), ArrDiff()`

ChanDiv()

This will divide the data from one or more channels by the data from another channel.

```
Func ChanDiv(cSpc, src%);
```

`cSpc` A channel list specifier of the channels to divide the source channel into. See the *Script language syntax* chapter for a definition of channel specifiers.

`src%` The number of the channel containing the data to divide into the other channels.

Returns Zero.

See also: `ChanAdd(), ChanSub(), ChanMult()`

ChanFit()

This function together with `ChanFitCoef()` and `ChanFitShow()` incorporates the functionality of the Analysis menu Fit Data dialog. `ChanFit()` has three variants, that initialise ready for a new fit, perform the fit and return information about the last fit. The current window must be a file, memory or XY view to use these functions.

Initialise fit information

This command associates a fit with a channel. The fit parameters and the coefficient limits are reset to their default values, the coefficient hold flags are cleared and any existing fit for this channel is removed.

```
Func ChanFit(chan%, type%, order%);
```

`chan%` The channel number to work on. Each channel in a file, memory or XY view can have one fit associated with it.

`type%` The fit type. 0=Clear any fit, 1=Exponential, 2=Polynomial, 3=Gaussian, 4=Sine, 5=Sigmoid.

`order%` The order of the fit. This can be 1 to 3 for a Sine or Gaussian fit, 1 for a Sigmoid fit, 1 to 5 for an exponential fit and 1 to 10 for a polynomial fit. If `type%` is 0 this should also be 0.

Returns 0 if the command succeeded.

Perform the fit

This variant of the command does the fit set by the previous variant.

```
Func ChanFit(chan%, opt%, frm%|frm%[]|frm$, start|start$,
             end|end${, ref|ref${, &err{, maxI% {,&iTer%{,
             covar[][]}}}}});
```

`chan%` A channel number in the current view that has had a fit initialised.

`opt%` This is the sum of:
1 Estimate the coefficients before fitting, else use current values.
2 Draw the fit over the user-defined range, not the fit range.

`frm%` Frame number or a negative code as follows:

- -1 All frames in the file.
- -2 The current frame.
- -3 Only tagged frames.
- -6 Only untagged frames.

`frm$` A frame specification string. This option specifies a list of frames using a string such as `"1..32,40,50"`.

`frm%[]` An array of frame numbers. This option provides a list of frame numbers. The first element holds the number of frames in the list.

`start` This is the start of the fit range, as a value or as a dialog expression string that is to be evaluated.

`end` The end of the fit range in x axis units, as a value or a string to evaluate.

`ref` The reference time as a value or a string to evaluate. If omitted `start` is used.

`err` If present, this optional variable is updated with the chi-squared or least squares error between the fit and the data.

`maxI%` If present, this sets the maximum number of iterations. If omitted, the current number set for the channel is used. The system default number is 100.

`iTer%` If present, this integer variable is updated with the count of iterations done.

`covar` An optional two dimensional array of size at least `[nCoef][nCoef]` that is returned holding the covariance matrix when the fit is complete. It is changed if the return value is –1, 0 or 1. However, the values it contains are probably not useful unless the return value is 0.

Returns 0 if the fit is complete, 1 if max iterations done, or a negative error code: -1=the fit is not making progress (results may be OK), -2=the fit failed due to a singular matrix, -5=the fit caused a floating point error, -6=too little data for the number of coefficients, -7=unknown fitting function, -8=ran out of memory during the fit (too many data points), -9=the fit was set up with bad parameters.

Get fit information

This variant of the command returns information about the current fit set for a channel.

```
Func ChanFit(chan%{, opt%});
```

`chan%` The channel number of the fit to return information about.

`opt%` This determines what information to return. If omitted, the default value is 0. Positive values return information about the fit that is set-up to be done next. Negative value return information about the last fit that was done and that can be displayed. The returned information for each value of `opt%` is:

opt%	Returns	opt%	Returns
0	Fit type of next fit	1	Fit order of next fit
-1	1=a fit exists, 0=no fit exists	-8	Reference x value
-2	Type of existing fit or 0	-9	User-defined x draw start
-3	Order of existing fit	-10	User-defined x draw end
-4	Chi or least-squares error	-11	1=chi-square, 0=least-square
-5	Fit probability (estimated)	-12	Last fit result code
-6	X axis value at fit start	-13	Number of fitted points
-7	X axis value at fit end	-14	Number of fit iterations used

Returns The information requested by the `opt%` argument or 0 if `opt%` is out of range.

See also: `ChanFitCoef()`, `ChanFitShow()`, `ChanFitValue()`, `FitExp()`, `FitPoly()`

ChanFitCoef()

This command gives you access to the fit coefficients for a channel in the current file, memory or XY view. You can return the values from any type of fit and set the initial values and limits and hold values fixed for iterative fits. There are two command variants:

Set and get coefficients

This command variant lets you read back the current coefficient values and set the coefficient values and limits for iterative fitting:

```
Func ChanFitCoef(chan%{, num%{, new{, lower{, upper}}}});
```

`chan%` The channel number of the fit to access.

`num%` If this is omitted, the return value is the number of coefficients in the current fit. If present, it is a coefficient number. The first coefficient is number 0. If `num%` is present, the return value is the coefficient value for the existing fit; if there is no fit, the coefficient value that would be used as the starting point for the next iterative fit is returned.

`new` If present, this sets the value of coefficient `num%` for the next iterative fit on this channel.

`lower` If present, this sets the lower limit for coefficient `num%` for the next iterative fit on this channel. There is currently no way to read back the coefficient limits. There is also no check made that the limits are set to sensible values.

`upper` If present, this sets the upper limit for coefficient `num%` for the next iterative fit on this channel.

Returns The number of coefficients or the value of coefficient `num%`.

Get and set the hold flags

This command variant sets the hold flags (equivalent to the Hold checkboxes in the Fit Data dialog Coefficients tab).

```
Func ChanFitCoef(chan%, hold%[]);
```

`chan%` The channel number of the fit to access.

`hold%` An array of integers to correspond with the coefficients. If the array is too long, extra elements are ignored. If it is too short, extra coefficients are not affected. Set `hold%[i%]` to 1 to hold coefficient `i%` and to 0 to fit it. If `hold%[i%]` is less than 0, the hold state is not changed, but `hold%[i%]` is set to 1 if the corresponding coefficient is held and to 0 if it is not held.

Returns This always returns 0.

See also: `ChanFit(), ChanFitShow(), ChanFitValue(), FitExp(), FitPoly()`

ChanFitShow()

This controls the display of data fitted to a channel in the current file, memory or XY view.

```
Func ChanFitShow(chan%{, opt%{, start|start${, end|end$}}});
```

`chan%` The channel number of the fit to access.

`opt%` If present and positive, this is the sum of:
1 Display the fitted data.
2 Use the user-defined display range rather than the fitting range.

If `opt%` is omitted or positive, the return value is the current option value. Use negative values to return the user-defined display range: -1=return the start, -2=return the end.

`start` If present, this is an x axis value or a string holding a dialog expression to be interpreted as an x axis value that sets the start of the user-defined display range.

`end` If present, this is an x axis value or a string holding a dialog expression to be interpreted as an x axis value that sets the end of the user-defined display range

Returns The current `opt%` value or the information requested by `opt%`. If there is no fit defined for the channel, the return value is 0.

See also: `ChanFit(), ChanFitShow(), ChanFitCoef(), FitExp(), FitPoly()`

ChanFitValue()

This function returns the value at a particular x axis value of the fitted function to a channel in the current file, memory or XY view.

```
Func ChanFitValue(chan%, x);
```

`chan%` The channel number of the fit to access.

`x` The x axis value at which to evaluate the current fit. You should be aware that some of the fitting fuctions can overflow the floating point range if you ask for x values beyond the fitted range of the function.

Returns The value of the fitted function at x. If the result is out of floating point range, the function may return a floating point infinity or a NaN (Not a Number) value or a 0. If there is no fit, the result is always 0.

See also: `ChanFit(), ChanFitShow(), ChanFitCoef(), FitExp(), FitPoly()`

ChanIntgl()

This integrates the data in specified waveform channels of the current frame in the current view. If the specified channel is not a waveform then the command has no effect. If the frame buffer is being shown, the frame buffer data is used instead. All data points in the channels used are modified.

```
Func ChanIntgl(cSpc);
```

`cSpc` A channel list specifier of the channels to integrate. See the *Script language syntax* chapter for a definition of channel specifiers.

`chan$` A string to specify channel numbers, such as `"1,3..8,9,11..16"`.

Returns 0 or a negative error code.

See also: `ShowBuffer(), ChanShow(), ChanZero(), ArrIntgl()`

ChanItems()

This counts waveform points, or markers, in a data view over an x axis range.

```
Func ChanItems(chan%, start, finish);
```

`chan%` The channel number (1 to n) for which to return information.

`start` The start position in x axis units. If start is greater than finish, the result is 0.

`finish` The last position in x axis units. If start equals finish, only items that fall exactly at the position count towards the result.

Returns The returned value is negative if the channel doesn't exist. Otherwise it is the number of data items in the range. This will be a count of markers or waveform points depending on the channel type.

See also: `ChanRange(), ChanPoints(), View(v,c).[], XYRange()`

ChanKind()

This returns the type of a channel in the current data or XY view.

```
Func ChanKind(chan%);
```

`chan%` The channel number (1 to n) for which to return information.

Returns -1 for a bad channel number, -2 if not a data or XY view, or:

0	Waveform	1	Marker	2	Text Marker (future)
3	XY channel	5	Idealised trace		

See also: `ViewKind()`

ChanList()

This function generates an array of channel numbers from the current data or XY view. The channels can be filtered to show only a subset of the available channels.

```
Func ChanList(list%[] {, types%});
Func ChanList(list%[], str${, types%});
```

`list%` An integer array to fill with channel numbers. Element 0 is set to the number of channels returned. The remaining elements are channel numbers. If the array is too short, enough channels are returned to fill the array. You will find that it is unnecessary to list all the channel numbers for an XY view, since they are numbered contiguously.

`types%` This argument specifies which channels to return. If omitted, all channels are returned. The values are the same as those defined for `mask%` with the `DlgChan()` function. This argument is ignored in an XY view where all channels are the same type.

`str$` A channel specification such as `"1..10,13,20"`. Only channels that exist in the current view are returned in `list%`. If `types%` is provided, only channels that match both the string and `types%` are returned in `list%`.

Returns The number of channels that would be returned if the array was of unlimited length or 0 if the view is not a data or XY view.

See also: `ChanShow()`, `ChanCount()`, `ChanDelete()`, `DlgChan()`, `XYSetChan()`

ChanMean()

This forms the mean level of a waveform channel in an x axis range.

```
Func ChanMean(chan%, start, finish[, stDev]);
```

`chan%` The channel number (1 to n) for which to form the mean.

`start` The start position in x axis units. If start is greater than finish, the result is 0.

`finish` The last position in x axis units.

`stDev` If present, this is returns the standard deviation of the data values in the range. If there is only one item the result is 0.

Returns It returns the sum of the data values in the range divided by the number of items. If the channel is not a waveform channel the script will fail.

See also: `ArrSum()`, `ChanMeasure()`

ChanMeasure()

This performs any of the cursor regions measurements on a channel.

```
Func ChanMeasure(chan%, type%, start|start$, end|end$);
```

`chan%` The channel number (1 to n) on which to perform the measurement.

`type%` The type of measurement to take, see the documentation of the Cursor Regions window for details of these measurements. The possible values are:

1 Curve area	2 Mean	3 Slope
4 Area	5 Sum	6 Modulus
7 Maximum	8 Minimum	9 Amplitude
10 RMS Amplitude	11 Standard dev.	12 Absolute maximum
13 Peak	14 Trough	

`start` The start position for the measurement in x axis units. If start is greater than finish, the result is 0.

`start$` The start position for the measurement expressed as a string. This allows constructs such as "`Cursor(1)`" to be used.

`end` The end position in x axis units.

`end$` The end position as a string. If the start value is a string, the end value must be a string as well.

Returns The function returns the requested measurement value.

See also: `ArrSum(), ChanMean(), ChanValue()`

ChanMult()

This will multiply the data in one or more channels by the data from another channel.

```
Func ChanMult(cSpc, src%);
```

`cSpc` A channel list specifier of the channels to multiply. See the *Script language syntax* chapter for a definition of channel specifiers.

`src%` The number of the channel containing the data to multiply the other chanels by.

Returns Zero.

See also: `ChanAdd(),ChanDiv(), ChanSub()`

ChanNegate()

This negates (inverts) the data in specified waveform channels of the current frame in the current view. If the frame buffer is being shown, the frame buffer data is used instead. All data points in the channels used are modified.

```
Func ChanNegate(cSpc);
```

`cSpc` A channel list specifier of the channels to negate. See the *Script language syntax* chapter for a definition of channel specifiers.

Returns 0 or a negative error code.

See also: `ShowBuffer(), ChanShow(), ChanZero(), ArrMul()`

ChanNumbers()

You can show and hide channel numbers in the current view and get the channel number state with this function. It is not an error to use this with data views that do not support channel number display, but the command has no effect.

```
Func ChanNumbers({show%});
```

`show%` If present, 0 hides the channel number, and 1 shows it. Other values are reserved (and currently have the same effect as 1).

Returns The channel number display state at the time of the call.

See also: `YAxis()`, `YAxisMode()`

ChanOffset()

This offsets the data in specified waveform channels of the current frame in the current view by adding a constant value to it. If the frame buffer is being shown, the frame buffer data is used instead. All data points in the channels used are modified.

```
Func ChanOffset(cSpc, val);
```

`cSpc` A channel list specifier of the channels to offset. See the *Script language syntax* chapter for a definition of channel specifiers.

val The value to add to the data, which can be negative for offsetting down.

Returns 0 or a negative error code.

See also: `ShowBuffer()`, `ChanScale()`, `ChanShift()`, `ChanZero()`, `ArrAdd()`

ChanOrder()

You can change the order of channels in a data or memory view, or group channels with a y axis so that they share a common axis with this command. This is equivalent to clicking and dragging the channel number. You can change individual channels, or sort all channels into numerical order:

```
Func ChanOrder(dest%, pos%, cSpc);
Func ChanOrder(dest%, opt%);
Func ChanOrder(order%);
```

`dest%` The destination channel number.

`Pos%` The position to drop the moved channels relative to the destination channel. Use –1 to drop before, 0 to drop on top and 1 to drop after. If you drop channels between grouped channels, then the dropped channels becomes members of the group (as long as they have a y axis).

`cSpc` A channel specifier for the channels to move.

opt% 0=returns the number of channels in the group that dest% belongs to or 0 if not grouped. 1-n returns the channel number of the nth channel in the group or 0 if no channel. -1 ungroups the group and returns the number of changed channels.

`order%` In this form of the command, all the channels are sorted into numerical order. Set –1 for low numbered channels at the top, 1 for High numbered channels at the top and 0 to use the default channel ordering set by the Edit menu Preferences.

Returns When used with a list of channels, the command returns the number of channels that were moved. When used to set the order of all channels, the return value is -1 if low numbered channels were placed at the top and 1 if high numbered channels were at the top.

See also: `ChanWeight()`, `ViewStandard()`

ChanPoints()

This function returns the total number of data items in the frame on the specified channel in a data or XY view.

```
Func ChanPoints(chan%);
```

`chan%` The channel number (1 to n) for which to return the number of items.

Returns The number of data points in a frame.

See also: `ChanRange(), ChanItems(), View(v,c).[], XYCount()`

ChanRange()

This function finds data items in a given x axis range.

```
Func ChanRange(chan%, &start, finish, &item);
```

`chan%` The channel number, from 1 to n.

`start` The start position in x axis units. This returns the start position of the first data point in the range, or it is left unchanged if no data is found.

`finish` The end position in x axis units.

`item` The index in the view of the data item found at the start position.

Returns The number of data items found in the range defined by `start` and `finish`.

See also: `ChanPoints(), ChanItems(), View(v,c).[]`

ChanRectify()

This rectifies the data in specified waveform channels of the current frame in the current view. If the frame buffer is being shown, the frame buffer data is used instead. All data points in the channels used are modified.

```
Func ChanRectify(cSpc);
```

`cSpc` A channel list specifier of the channels to rectify. See the *Script language syntax* chapter for a definition of channel specifiers.

Returns 0 or a negative error code.

See also: `ShowBuffer(), ChanShow(), ChanVisible(), ChanZero()`

ChanScale()

This scales the data in specifed waveform channels of the current frame in the current view by multiplying it by a constant value. If the frame buffer is being shown, the frame buffer data is used instead. All data points in the channels used are modified.

```
Func ChanScale(cSpc, val);
```

`cSpc` A channel list specifier of the channels to scale. See the *Script language syntax* chapter for a definition of channel specifiers.

`val` The value to multiply by.

Returns 0 or a negative error code.

See also: `ShowBuffer(), ChanShow(), ChanZero(), ArrMul()`

ChanSearch()

This function searches a channel in file or memory view for the next user-defined feature in a time range. It is exactly the same as an active cursor search, but does not use or move cursors. By avoiding the need to draw and move cursors, this function should be more efficient than using the active cursors, but there is no visual feedback.

```
Func ChanSearch(chan% ,mode%, &sT, eT{, sp1{, sp2{, width}}});
```

`chan%` The number of a channel in the file or memory view to search.

`mode%` This sets the search mode, as for the active cursors. See the cursor mode dialog documentation for details of each mode.

1 Maximum value	7 Rising threshold	13 Slope threshold
2 Minimum value	8 Falling threshold	14 Rising slope threshold
3 Maximum excursion	9 Steepest rising	15 Falling slope threshold
4 Peak find	10 Steepest falling	16 Absolute max slope
5 Trough find	11 Slope peak	17 Turning point
6 Threshold	12 Slope trough	18 Slope%

`sT` The start time for the search. This value will be returned with the result of the search.

`eT` The end time for the search. If `eT` is less than `sT`, the search is backwards.

`sp1` This is the threshold level for threshold crossings and baseline level for maximum excursion. It is in the y axis units of the search channel (y axis units per second for slopes). If omitted, the value 0.0 is used. Set it to 0 if `sp1` is not required for the mode.

`sp2` This is hysteresis for peak, trough and threshold crossings and percent for Slope%. If omitted, the value 0.0 is used. Set it to 0 if `sp2` is not required for the mode.

`width` This is the width in seconds for all slope measurements. If omitted, the value 0.0 is used. Set it to 0 if `width` is not required for the mode.

Returns 0 if the search succeeds or -1 if the search fails or a negative error code.

See also: `ChanMeasure(), ChanValue(), CursorActiveGet(), CursorActiveSet()`

ChanSelect()

This function is used to report on the selected/unselected state of a channel in a data view, and to change the selected state of a channel.

```
Func ChanSelect(cSpc {,new%});
```

`cSpc` A channel list specifier of the channels to select. See the *Script language syntax* chapter for a definition of channel specifiers.

`new%` If present it sets the state: 0 for unselected, not 0 for selected. If omitted, the state is unchanged. Attempts to change invisible channels are ignored.

Returns If you set `chan%` to a positive channel number the function returns the channel state at the time of the call, 0 for unselected, 1 for selected. Otherwise the function returns the number of selected channels at the time of the call.

See also: `ChanList(), ChanOrder(), ChanVisible(), ChanWeight()`

ChanShift()

This shifts the data in specified waveform channels of the current frame in the current view a specified number of points right or left. The data is actually rotated so that points that 'fall off' one end are shifted back in at the other. If the frame buffer is being shown, the frame buffer data is used instead. All data points in the channels used are modified.

```
Func ChanShift(cSpc, shift%);
```

`cSpc` A channel list specifier of the channels to shift. See the *Script language syntax* chapter for a definition of channel specifiers.

`shift%` The number of point to shift the data. A negative value shifts points left, a positive value shifts points right.

Returns 0 or a negative error code.

See also: `ShowBuffer(), ChanShow(), ChanZero(), ArrAdd()`

ChanShow()

Display or hide a channel, or a list of channels, in a data or XY view. Turning on a channel that is on has no effect. Turning on a channel that doesn't exist has no effect.

```
Func ChanShow(cSpc {,yes%});
```

`cSpc` A channel list specifier of the channels to display or hide. See the *Script language syntax* chapter for a definition of channel specifiers.

`yes%` If this is non-zero it turns the specified channels on and if it is zero it turns them off. If `yes%` is omitted no changes are made.

Returns If you set `chan%` to a positive channel number the function returns the channel state at the time of the call, 1 for visible, 0 for invisible. Otherwise it returns -1.

See also: `ChanList(), ChanVisible()`

ChanSmooth()

This smooths the data in specified waveform channels of the current frame in the current view by replacing each point by the average of n adjacent points, where n can be 3 or 5. If the frame buffer is being shown, the frame buffer data is used instead. All data points in the channels used are modified.

```
Func ChanSmooth(cSpc, width);
```

`cSpc` A channel list specifier of the channels to smooth. See the *Script language syntax* chapter for a definition of channel specifiers.

`width` This sets the width to smooth over, either 3 or 5.

Returns 0 or a negative error code.

See also: `ShowBuffer(), ChanShow(), ChanZero(), ArrFilt()`

ChanSub()

This will subtract the data in one channel from the data in one or more other channels.

```
Func ChanSub(cSpc, src%);
```

`cSpc` A channel list specifier of the channels to subtract from. See the *Script language syntax* chapter for a definition of channel specifiers.

`src%` The number of the channel containing the data to subtract.

Returns Zero.

See also: `ChanAdd(),ChanDiv(), ChanMult()`

ChanSubDC()

This function subtracts any DC offset present in the data in specifed waveform channels of the current frame in the current view. The DC offset is measured over the time range specified, all data points in the channels are modified. If the frame buffer is being shown, the frame buffer data is used instead.

```
Func ChanSubDC(cSpc, start, finish);
```

`cSpc` A channel list specifier of the channels to subtract from. See the *Script language syntax* chapter for a definition of channel specifiers.

`start` The start position for measurement of the DC level.

`finish` The end position for measurement of the DC level.

Returns 0 or a negative error code.

See also: `ShowBuffer(), ChanShow(), ChanZero(), ArrSum()`

ChanTitle$()

This returns the title for a channel in a data or XY view. In a memory or XY views, or in a sampling document view, it can also set the channel title. For XY views only, it can also be used to get or set the Y axis title. In an XY view the channel titles are visible in the Key window.

```
Func ChanTitle$(chan%{,new$});
```

`chan%` The channel number (1 to n). For an XY view only, a channel number of zero can be used to access the Y axis title.

`new$` If present, in a sampling document or memory view, this string holds the new channel title. If the string is too long, it is truncated.

Returns The original title for the channel. If the channel does not exist, the function does nothing and returns an empty string

See also: `ChanUnits$(), XTitle$(), XYKey(), YAxis()`

ChanUnits$()

This returns the units for a waveform channel in a data view or the Y axis units in an XY view. In a memory, XY, or sampling document view, it can also set the units.

```
Func ChanUnits$(chan%{,new$});
```

`chan%` A channel number (1 to n). This is ignored in an XY, where it operates on the Y axis units only.

`new$` If present, in a sampling document, XY or memory view, this string holds the new channel title. If the string is too long, it is truncated.

Returns It returns the original units for the channel. If the channel does not exist or is not of a suitable type, the function does nothing and returns an empty string.

See also: `ChanTitle$(), XUnits$(), YAxis()`

ChanValue()

This returns the value on a given channel at a given position. It returns a value in the y axis units of the channel display mode. If the display mode has no y axis the value is the x axis position of the next item on the channel.

This returns the value corresponding to an x axis value. Use the `View(v,c).[bin]` notation or `BinToX(bin)` to access view data by bin number.

```
Func ChanValue(chan%, pos {,&data%{,mode%{,binsz}}});
```

`chan%` A channel number (1 to n).

`pos` The x axis position for which the value is needed.

`data%` This is returned as 1 if there was data at the position, 0 if there was not. For example on a waveform channel with time on the x axis, if there was no waveform point within `BinSize(chan%)` of the time set by `pos`, this would be set to 0.

`mode%` This will have no effect for a waveform channel. If present for a marker channel, this sets the display mode to use for extracting a value from a view. If an inappropriate mode is requested or if `mode%` is absent, the actual display mode is used. The modes are:

0 The current mode for the channel. Any additional arguments are ignored.
1 Dots mode for markers, returns the position of the marker at or after `pos`.
2 Lines mode for markers, result is the same as mode 1.
3 Rate mode for markers. If the `binSz` argument is present it sets the width of each bin, otherwise the bin width is set to 1.0.

`binSz` If present, when `mode%` specifies rate mode for markers, this sets the width of the rate histogram bins in x axis units.

Returns It returns the value or zero if no data is found. For display modes with a y axis, if there is no data within `BinSize(chan%)` of the position, the value is zero. This is the same value returned by the Cursor Values menu for the channel.

If `data%` is not provided, any error stops the script. Errors include: no current window, current window not a data view, no data at `pos`, and `pos` beyond range of x axis. If `data%` is present, errors cause `data%` to be set to 0.

For example, to get data value on channel 1 at the position of the cursor number 1 in the view on data file, `mydata.cfs`:

```
vdata%:=ViewFind("mydata.cfs");   'view handle for data
FrontView(vdata%);                'focus on the data window
ampl:=ChanValue(1,Cursor(1));     'get data value at cursor
```

See also: `BinToX(), Cursor(), ChanMeasure(), DrawMode(), Interact(), View(v,c).[]`

ChanVisible()

This returns the show state of the channel as 1 if the channel is visible and 0 if it is not. If you use a silly channel number, the result is 0 (not displayed).

```
Func ChanVisible(chan%);
```

`chan%` The channel number (1 to n) to report on.

Returns 1 if the channel is displayed, 0 if it is not.

See also: `ChanShow()`

ChanWeight()

This function sets the relative vertical space to give a channel or a list of channels. The standard vertical space corresponds to a weight of 1. When Signal allocates vertical space, channels are of two types: channels with a y axis and channels without a y axis. Signal calculates how much space to give each channel type assuming all channels have a weight of 1. Then the actual space allocated is proportional to the standard space multiplied by the weight factor. This means that if you increase the weight of one channel, all other channels get less space in proportion to their original space.

```
Func ChanWeight(cSpc{, new});
```

`cSpc` The specification for the list of channels to process. See the *Script language syntax* chapter for a definition of channel specifiers.

`new` If present, a value between 0.001 and 1000.0 that sets the weight for all the channels in the list. Values outside this range are limited to the range.

Returns The command returns the channel weight of the first channel in the list.

See also: `ChanOrder()`, `ViewStandard()`

ChanZero()

This sets to zero the data in specified waveform channels of the current frame in the current view. If the frame buffer is being shown, the frame buffer data is used instead. All data points in the channels used are modified.

```
Func ChanZero(cSpc);
```

`cSpc` A channel list specifier of the channels to zero. See the *Script language syntax* chapter for a definition of channel specifiers.

Returns 0 or a negative error code.

See also: `ShowBuffer()`, `ChanShow()`, `ArrConst()`

Chr$()

This function converts a code to a character and returns it as a single character string.

```
Func Chr$(code%);
```

`code%` The code to convert. Codes that have no character representation will produce unpredictable results when printed or displayed.

See also: `Asc()`, `DelStr$()`, `LCase$()`, `Left$()`, `Len()`, `Mid$()`, `Print$()`, `Right$()`, `Str$()`, `UCase$()`, `Val()`

Colour()

This function gets and/or sets the colours of items. Colours are set in terms of the colour palette displayed in the Colour menu, not directly in terms of colours. XY view channels are coloured using `XYColour()`.

```
Func Colour(item% {,col%});
```

`item%` This is the item number, being the position of the item in the Colour menu, as follows:

1	Data view background	11	Rate histogram fill
2	Waveform as line	12	Text labels
3	Waveform as dots	13	Cursors & cursor labels
4	Waveform as skyline	14	Controls (not used)
5	Waveform as histogram	15	Data display grid
6	Waveform histogram fill	16	Axis markings and text labels
7	Markers as dots	17	Tagged frames background
8	Makers as lines	18	Frame list traces
9	Marker text	19	XY view background
10	Rate histogram	20	Error bar colour

`col%` If present, this sets the index of the colour in the colour palette to be applied to the item. There are 40 colours in the palette, numbered 0 to 39. The first 7 colours in the palette are set to grey scales from black to white, and the rest can be selected or mixed from basic colours.

Returns The index into the colour palette of the colour of the item at the time of the call.

See also: `PaletteGet(), PaletteSet(), XYColour(), ChanColour()`

CondXXX() Conditioner commands

The `Cond…` family of commands control external signal conditioners. At the time of writing, these commands support the CED 1902 programmable signal conditioner, the Power1401 programmable gain option and the Axon Instruments CyberAmp. Other conditioners may be added in the future.

These commands do not define which serial port is used by the conditioner nor the type of conditioner supported. When you install Signal you must choose the conditioner type and set the serial port. All these commands require a `port%` argument. This is the physical waveform input port number that the conditioner channel is attached to. It is not a channel number in a view.

You can access the built-in interactive support for the conditioner from the sampling configuration channel setup dialog. This can be a useful short-cut to getting the lists of gains and signal sources available on your conditioner(s).

See also: `CondFilter()`, `CondFilterList()`, `CondGain()`, `CondGainList()`, `CondGet()`, `CondOffset()`, `CondOffsetLimit()`, `CondRevision$()`, `CondSet()`, `CondSourceList()`, `CondType()`

CondFeature()

This command gets and sets special signal conditioner features that are not general enough to have dedicated commands to support them. See the `CondSet()` command for more details of conditioner operation. There are four command variants:

Get feature count

```
Func CondFeature(port%);
```

`port%` The waveform port number that the conditioner is connected to.

Returns The number of special features supported by the signal conditioner.

Get feature information

The command supports two types of features: those that have a set of discrete values such as ["None", "Rectify"], and those that support a continuous range of floating point values, such as 10.0 to 25.6, for example. To determine if the feature is continuous or discrete, call the function with only the first 3 or 4 arguments.

```
Func CondFeature(port%, feat%, &name${,&flags%{, list$[]}});
Func CondFeature(port%, feat%, &name${,&flags%{, &low{, &high}}});
```

`feat%` The feature number, from 1 to the number of features available

`name$` Returned set to the name of the feature.

`flags%` Returned set to the feature flags. Currently, none are defined, so this will be 0.

`list$` Returned set to an array of the possible settings (strings) for discrete type.

`low` Returned as the low limit for a feature with continuous values.

`high` Returned as the upper limit for a feature with continuous values

Returns The number of discrete feature values, or 0 if the feature supports continuous values over the range returned in `low` and `high`.

Set feature value

```
Func CondFeature(port%, feat%{, val});
```

`val` If present, it sets the value for the feature set by `feat%` and `port%`. If this is a continuous feature, `val` sets the new value. If the feature has `n` discrete setting, `val` should be 0 to `n`-1 to select the feature corresponding to the feature description in `list$[val]`. If `val` exceeds the allowed range, a continuous feature is set to the nearest allowed value and a discrete feature is unchanged.

Returns The feature value at the time of the call (before any change). This is an integer index for features with discrete values otherwise it is the feature value.

See also: `CondFilterList()`, `CondGain()`, `CondGainList()`, `CondGet()`, `CondSet()`, `CondSourceList()`, `CondType()`

CondFilter()

This sets or gets the frequency of the low-pass or high-pass filter of the signal conditioner. See the `CondSet()` command for more details of conditioner operation.

```
Func CondFilter(port%, high% {,freq{, type%}});
```

`port%` The waveform port number that the conditioner is connected to.

`high%` This selects which filter to set or get: 0 for low-pass, 1 for high-pass.

`freq` If present, this sets the desired corner frequency of the selected filter. See the `CondSet()` description for more information. Set 0 for no filtering. If omitted, the frequency is not changed. The high-pass frequency must be set lower than the frequency of the low-pass filter, if not the function returns a negative code.

`type%` Optional, taken as 1 if omitted. The filter type to use when setting the filter. The filter type in the range 1 to the number of types (as returned by `CondFilterType()`).

Returns The cut-off frequency of the selected filter at the time of call, or a negative error code. A return value of 0 means that there is no filtering of the type selected.

See also: `CondFilterList()`, `CondFilterType()`, `CondGain()`, `CondGainList()`, `CondGet()`, `CondOffset()`, `CondOffsetLimit()`, `CondRevision$()`, `CondSet()`, `CondSourceList()`, `CondType()`

CondFilterList()

This function gets a list of the possible filter frequencies of the conditioner. Conditioners that support continuous frequency ranges also supply a list of frequencies to match the list of frequencies shown in the conditioner control panel. See the `CondSet()` command for more details of conditioner operation.

```
Func CondFilterList(port%, high%, freq[]{, type%});
```

`port%` The waveform port number that the conditioner is connected to.

`high%` Selects which filter to get: 0 for low-pass, 1 for high-pass.

`freq[]` An array of reals holding the cut-off frequencies of the selected filter. There is always a value of 0 meaning no filtering.

`type%` Optional, taken as 1 if omitted. The filter type in the range 1 to the number of types (as returned by `CondFilterType()`).

Returns The number of filter frequencies (including 0) or a negative error code.

See also: `CondFilter()`, `CondFilterType()`, `CondGet()`, `CondOffset()`, `CondOffsetLimit()`, `CondSet()`, `CondSourceList()`, `CondType()`

CondFilterType()

This function returns information about the filter types supported by the conditioner for the low pass and high pass filters. For example, the CED 1902 mk IV supports a choice of filter types. There are three command variants:

Get number of filter types

```
Func CondFilterType(port%, high%);
```

`port%` The waveform port number that the conditioner is connected to.

`high%` Selects which filter to return information for: 0 for low-pass, 1 for high-pass.

Returns The number of filter types.

Get filter type in use

```
Func CondFilterType(port%, high%, 0);
```

Returns The currently selected filter type, from 1 to the number of filter types.

Get filter type information

```
Func CondFilterType(port%,high%,type%{,&name${,&lower{,&upper}}});
```

`type%` The filter number, from 1 to the number of available filters.

`name$` If present, returned holding the name of the filter selected by `type%`.

`lower` If present, returned as the lowest filter frequency (excluding 0, meaning 'off').

`upper` If present, returned as the highest supported filter frequency.

Returns The number of frequency values the filter can be set to (including 0) or 0 if the filter corner frequency can be set to any value in the range `lower` to `upper`.

`See also: CondFilter(), CondGain(), CondGainList(), CondGet(), CondSet(), CondSourceList(), CondType()`

CondGain()

This sets and gets the gain of the signal passing through the signal conditioner See the `CondSet()` command for more details of conditioner operation.

```
Func CondGain(port% {,gain});
```

`port%` The waveform port number that the conditioner is connected to.

`gain` If present this sets the ratio of output signal to the input signal. If this argument is omitted, the current gain is returned. The conditioner will set the nearest gain it can to the requested value.

Returns The gain at the time of call, or a negative error code.

See also: `CondFilter()`, `CondFilterList()`, `CondGainList()`, `CondGet()`, `CondOffset()`, `CondOffsetLimit()`, `CondSet()`

CondGainList()

This function gets a list of the possible gains of the conditioner for the selected signal source. See the `CondSet()` command for more details of conditioner operation.

```
Func CondGainList(port%, gain[]);
```

`port%` The waveform port number that the conditioner is connected to.

`gain[]` An array of reals holding the conditioner gains for the selected signal source. If a conditioner (for example, 1902) has a fixed set of gains, this is the set of gain values. If the conditioner supports continuously variable gain, the first two elements of this array hold the minimum and the maximum values of the gain.

Returns The number of gain values if the conditioner has a fixed set of gains or 2 if the conditioner has continuously variable gain. In the case of an error, a negative error code is returned.

See also: `CondFilter()`, `CondFilterList()`, `CondGain()`, `CondOffset()`, `CondOffsetLimit()`, `CondSet()`, `CondSourceList()`

CondGet()

This function gets the input signal source of the signal conditioner, and the conditioner settings for gain, offset, filters and coupling. The settings are returned in arguments which must all be variables. See `CondSet()` for details of conditioner operation.

```
Func CondGet(port%, &in%, &gain, &offs, &low, &hi, &notch%, &ac%
                                        {, &typeL%{, &typeH%}});
```

`port%` The waveform port number that the conditioner is connected to.

`in%` Returned as the zero-based index of the input signal source (see `CondSet()`).

`gain` Returned as the ratio of output to input signal amplitude (ignoring filtering).

`offs` A value added to the input waveform to move it into a more useful range. Offset is specified in user units and is only meaningful when DC coupling is used.

`low` Returned as the cut-off frequency of the low-pass filter. A value of 0 means that there is no low-pass filtering enabled on this channel.

`hi` Returned as the cut-off frequency of the high-pass filter. A value of 0 means that there is no high-pass filtering enabled on this channel.

`notch%` Returned as 0 if the mains notch filter is off, and 1 if it is on.

`ac%` Returned as 1 for AC or 0 for DC coupling.

`typeL%` Optional integer variable returned holding the low-pass filter type number as described for `CondFilterType()`.

`typeH%` Optional integer variable returned holding the high-pass filter type number.

Returns 0 if all well or a negative error code.

See also: `CondFilter()`, `CondFilterType()`, `CondFilterList()`, `CondGain()`, `CondGainList()`, `CondOffset()`, `CondSet()`, `CondSourceList()`

CondOffset()

This sets or gets the offset added to the input signal of the signal conditioner. See the `CondSet()` command for more details of conditioner operation.

```
Func CondOffset(port%{, offs});
```

`port%` The waveform port number that the conditioner is connected to.

`offs` The value to add to the input waveform of the conditioner to move it into a more useful range. If this argument is omitted, the current offset is returned. The conditioner will set the nearest value it can to the requested value.

Returns The offset at the time of call, or a negative error code.

See also: `CondGain()`, `CondGainList()`, `CondGet()`, `CondOffsetLimit()`, `CondRevision$()`, `CondSet()`, `CondSourceList()`, `CondType()`

CondOffsetLimit()

This function gets the maximum and minimum values of the offset range of the conditioner for the currently selected signal source.

```
Func CondOffsetLimit(port%, offs[]);
```

`port%` The waveform port number that the conditioner is connected to.

`offs[]` This is an array of real numbers returned holding the minimum (`offs[0]`) and the maximum (`offs[1]`) values of the offset range of the conditioner for the currently selected signal source.

Returns 2 or a negative error code.

See also: `CondGain()`, `CondGainList()`, `CondGet()`, `CondOffset()`, `CondRevision$()`, `CondSet()`, `CondSourceList()`, `CondType()`

CondRevision$()

This function returns the name and version of the signal conditioner as a string or an empty string if there is no conditioner for the port.

```
Func CondRevision$(port%);
```

`port%` The waveform port number that the conditioner is connected to.

Returns A string describing the conditioner. Strings defined so far include: "`1902ssh`", where `ss` is the 1902 ROM software version number and `h` is the hardware revision level; and "`CYBERAMP 3n0 REV x.y.z`" where `n` is 2 or 8.

See also: `CondFeature()`, `CondFilter()`, `CondType()`

CondSet()

This sets the input signal source, gain, offset, filters and coupling of the conditioner. All values are requests; the command sets the closest possible value to that requested. If it is important to know what has actually been set you should read back the values with `CondGet()` after setting them, or use the functions for reading specific values.

```
Func CondSet(port%, in%, gain, offs {,low, high, notch%, ac%
                                        {, typeL%{, typeH%}}});
```

`port%` The waveform port number that the conditioner is connected to.

`in%` A conditioner has one or more signal sources. For example, the CED 1902 supports `Grounded`, `Single ended`, `Normal Diff`, `Inverted Diff`, etc. Conditioners of the same type may have different sources. To select a source, set `in%` to its zero-based index in the list returned by `CondSourceList()`.

`gain` This is the desired ratio of output to the input signal amplitude (ignoring the effect of any filtering). The actual gain depends on the capabilities of the signal conditioner, see `CondGainList()`. The gain range may be altered by the choice of signal source. For example, the 1902 Isolated Amp input has a build-in gain of 100. This command sets the nearest gain to the requested value.

`offs` This is the desired value in user units to add to the input waveform to move it into a more useful range. Offsets are only meaningful with DC coupling. Different conditioners have different offset ranges, and the offset range may be altered by the choice of signal source, see `CondOffsetLimit()`. The command will set the nearest offset it can to the desired value.

`low` If present and greater than 0, it is the desired corner frequency of the low-pass filter. Low-pass filters are used to reduce the high frequency content of the signal, both to satisfy the sampling requirement, and in case where it is known that no useful information is to be found in the signal above a certain frequency. If omitted, or 0, there is no low-pass filtering. The actual filter value set depends on the capabilities of the signal conditioner.

`high` If present and greater than 0, it is the high-pass filter corner frequency. High-pass filters reduce the low-frequency content of the signal. This must be set lower than the frequency of the low-pass filter; if not, the function returns a negative code. If omitted, or set to 0, there is no high-pass filtering.

Different signal conditioners have different ranges of frequency filtering. To find out the real filter frequency set, use `CondFilter()`. `CondFilterList()` returns the list of possible filter frequencies.

`notch%` Some signal conditioners have a mains-frequency notch filter (usually 50 Hz or 60 Hz) used to reduce the effect of mains interference on low level signals. This filter will remove the fundamental 50 Hz or 60 Hz signal; it will not remove higher harmonics (for example 150 Hz). If `notch%` is present with a value greater than 0, the notch filter is on. If omitted, or 0, the notch filter is off.

`ac%` The 1902 supports both AC and DC signal coupling. If you set AC coupling you should probably set the offset to zero. If `ac%` is greater than 0, the signal conditioner is AC coupled. If omitted or 0, the signal conditioner is DC coupled.

`typeL%` Optional value, taken as 1 if omitted, that sets the low-pass filter type as described for `CondFilterType()` in the range 1 to the number of filter types.

`typeH%` Optional value, taken as 1 if omitted, that sets the high-pass filter type.

Returns 0 if all well or a negative error code.

See also: `CondFilter()`, `CondFilterType()`, `CondGain()`, `CondGet()`, `CondOffset()`, `CondSourceList()`

CondSourceList()

This function gets a list of the possible signal source names of the conditioner, or the specific signal source name with the given index number.

```
Func CondSourceList(port%, src$[]|src$ {,in%});
```

`port%` The waveform port number that the conditioner is connected to.

`src$` This is either a string variable or an array of strings that is returned holding the name(s) of signal sources. Only one name is returned per string.

`in%` This argument lets you select an individual source or all sources. If present and greater than or equal to 0, it is the zero-based index number of the signal source to return. In this case, only one source is returned, even if `src$` is an array.

If omitted and `src$` is a string, the first source is returned in `src$`. If `src$[]` is an array of strings, as many sources as will fit in the string array are returned.

Returns If `in%` is greater than or equal to 0, it returns 1 or a negative error code. If `in%` is omitted, it returns the number of signal sources or a negative error code.

See also: `CondFilterList()`, `CondGainList()`, `CondGet()`, `CondOffsetLimit()`, `CondRevision$()`, `CondType()`

CondType()

This function returns the type of the signal conditioner.

```
Func CondType(port%);
```

`port%` The waveform port number that the conditioner is connected to.

Returns 0 for no conditioner or it is not the type set when installing, 1 for a CED 1902, 2 for an Axon Instruments CyberAmp and 3 for Power1401 with gain controls.

See also: `CondFilterList()`, `CondGainList()`, `CondRevision$()`, `CondSourceList()`

Cos()

This calculates the cosine of one or an array of angles in radians.

```
Func Cos(x|x[]{[]...});
```

`x` The angle, expressed in radians, or a real array of angles. The best accuracy of the result is obtained when the angle is in the range -2π to 2π.

Returns When the argument is an array, the function replaces the array with the cosines of all the points and returns either a negative error code or 0 if all was well. When the argument is not an array the function returns the cosine of the angle.

See also: `ATan()`, `Cosh()`, `Ln()`, `Log()`, `Pow()`, `Sin()`, `Sinh()`, `Sqrt()`, `Tan()`

Cosh()

This calculates the hyperbolic cosine of one value or an array of values.

```
Func Cosh(x|x[]{[]...});
```

`x` The value, or a real array of values.

Returns When the argument is an array, the function replaces each value with its hyperbolic cosine and returns 0. When the argument is not an array the function returns the cosh of the argument.

See also: `ATan()`, `Cos()`, `Exp()`, `Ln()`, `Log()`, `Pow()`, `Sinh()`, `Sqrt()`, `Tanh()`

Cursor()

This function returns the x axis position of a vertical cursor, and can also move the cursor to a new position.

```
Func Cursor(num% {,where});
```

`num%` The cursor number to use.

`where` If present, the new position of the cursor. If the new position is out of range of the x axis, it is limited to the x axis.

Returns The old cursor position or 0 if the cursor doesn't exist.

Examples:

```
Cursor(1,2.0);              'Set cursor 1 at position 2.0
where := Cursor(1);         'Get cursor position
```

See also: `ChanValue()`, `CursorDelete()`, `CursorLabel()`, `CursorLabelPos()`, `CursorNew()`, `CursorRenumber()`, `CursorSet()`, `HCursor()`

CursorActiveGet()

This command returns the parameters used by an active cursor in searching for a feature in the view data. Note that the use of some parameters varies according to the cursor mode set. You should check the cursor mode first since if it is Repolarisation % then the fifth argument should be a string and not a real variable.

```
Func CursorActiveGet(num%, chan%, start|start$, end|end$
                  {, thresh|ref${, hyst {, width{, def|def$}}}});
```

`num%` The cursor number.

`chan%` Returned holding the number of thr channel on which the cursor operates.

`start` Returned holding the start time for the feature search.

`start$` Returned holding the start time for the search string or the expression used in mode 20. Search limits such as `"XLow() + 0.2"` can be correctly returned.

`end` Returned holding the end time for the search.

`end$` Returned holding the end time for the search as a string. If the string form of `start` is used, the string form of `end` must also be used, and vice versa.

`thresh` Returned holding the threshold level used in the feature search.

`ref$` Returned holding the expression used in mode 19 (Repolarisation %) to define the time at which the 100% value is measured. The 0% value is measured at the start time.

`hyst` Returned holding the hysteresis value used in the feature search or the % value for modes 18 (percentage slope) and 19 (Repolarisation %)

`width` Returned holding the slope width value used in the feature search.

`def` Returned holding the default position if the search fails

`def$` Returned holding the default position as a string.

Returns Zero.

See also: `Cursor()`, `CursorDelete()`, `CursorMode()`, `CursorNew()`, `CursorActiveSet()`

CursorActiveSet()

This command sets the parameters used by an active cursor in searching for a feature in the view data. Note that the use of some parameters varies according to the cursor mode set.

```
Func CursorActiveSet(num%, chan%, start|start$, end|end$ {,
                     thresh|ref${, hyst {, width{, def|def$}}}});
```

`num%` The cursor number.

`chan%` The number of the channel on which the cursor operates.

`start` The start time for the feature search.

`start$` The start time for the search expressed as a string or the expression used in mode 20. Expressions such as `"XLow() + 0.2"` can be used.

`end` The end time for the search.

`end$` The end time for the search as a string. If the string form of `start` is used, the string form of `end` must also be used, and vice versa.

`thresh` The threshold level used in the feature search.

`ref$` The expression used in mode 19 (Repolarisation %) to define the time at which the 100% value is measured. The 0% value is measured at the start time.

`hyst` The hysteresis value used in the feature search or the % value for modes 18 (percentage slope) and 19 (Repolarisation %).

`width` The slope width value used in the feature search.

`def` The default position if the search fails.

`def$` The default position as a string.

Returns Zero.

See also: `Cursor()`, `CursorDelete()`, `CursorMode()`, `CursorNew()`, `CursorActiveGet()`

CursorDelete()

Deletes a cursor. It is not an error to delete an unknown cursor (which has no effect).

```
Func CursorDelete({num%});
```

`num%` The cursor number to delete, or -1 to delete all cursors. If omitted, the highest-numbered cursor is deleted.

Returns The number of the cursor deleted or 0 if no cursor was deleted.

See also: `Cursor()`, `CursorLabel()`, `CursorLabelPos()`, `CursorNew()`, `CursorRenumber()`, `CursorSet()`, `HCursorDelete()`

CursorExists()

This function tests if a given vertical cursor exists at the time of the call.

```
Func CursorExists(num%);
```

`num%` The cursor number.

Returns 1 if the cursor exists, 0 if it does not.

See also: `Cursor()`, `CursorDelete()`, `HCursorExists()`

CursorLabel()

This command sets (or gets) the cursor label style for the current view. Cursors can be annotated with a position and/or the cursor number, or with a user-defined string:

```
Func CursorLabel({style%{, num%{, form$}}});
```

`style%` Label styles are: 0=None, 1=Position, 2=Number, 3=Both, 4=User-defined. Unknown styles cause no change. Style 4 is used with a format string.

`num%` Used with style 4 only. A value of 0 selects all cursors, 1 upwards selects one cursor.

`form$` A string to label the cursors. It has replaceable parameters `%p`, `%n` and `%v(chan)` for position, number and channel value (replace `chan` with the channel number whose value you require). `%w.dp` and `%w.dv(chan)` formats are allowed where `w` and `d` are numbers that set the field width and number of decimal places.

Returns The current style of cursor 1 before any change. If `style%` is omitted, the current cursor style is not changed.

See also: `Cursor(), CursorDelete(), CursorLabelPos(), CursorNew(), CursorRenumber(), CursorSet(), HCursorLabel()`

CursorLabelPos()

This lets you set and read the position of the cursor label.

```
Func CursorLabelPos(num% {,pos});
```

`num%` The cursor number. If the cursor does not exist the function does nothing and returns -1.

`pos` If present, the command sets the label position as the percentage of the distance from the top of the cursor. Out-of-range values are set to the appropriate limit.

Returns The cursor position before any change was made, or -1 if the cursor does not exist.

See also: `Cursor(), CursorDelete(), CursorLabel(), CursorNew(), CursorRenumber(), CursorSet(), HCursorLabelPos()`

CursorMode()

This lets you set and read the a cursor mode.

```
Func CursorMode(num% {,mode%});
```

`num%` The cursor number. If the cursor does not exist the function does nothing and returns -1.

`Mode%` If present, the command sets the new cursor mode (see the main active cursor documentation for a description of the modes). The possible values of `mode%` are:

0 Static	1 Maximum	2 Minimum
3 Maximum excursion	4 Peak	5 Trough
6 Threshold	7 +ve threshold	8 -ve threshold
9 Max slope	10 Min slope	11 Peak in slope
12 Trough in slope	13 Slope threshold	14 Slope +ve thresh.
15 Slope -ve thresh.	16 Abs max slope	17 Turning point
18 percentage slope	19 Repolarisation %	20 Expression

Returns The cursor mode before any change was made, or -1 if the cursor does not exist.

See also: `Cursor(), CursorActiveGet(), CursorActiveSet(), CursorNew(), CursorSet()`

CursorNew()

This command adds a new cursor to the view at the designated position. A new cursor is created in Static mode (not active).

```
Func CursorNew({where{, num%}});
```

`where` Where to position the cursor. In a file or memory view it is a time in seconds. In an XY view it is in x axis units. The position is limited to the x axis range. If the position is omitted, the cursor is placed in the middle of the window.

`num%` If this is omitted, or set to -1, the lowest numbered free cursor is used. If this is a cursor number, that cursor is created. This must be a legal cursor number or -1.

Returns It returns the cursor number as an integer, or 0 if all cursors are in use.

See also: `Cursor(), CursorDelete(), CursorLabel(), CursorLabelPos(), CursorRenumber(), CursorSet(), HCursorNew()`

CursorRenumber()

This command renumbers the cursors from left to right in the view. There are no arguments.

```
Func CursorRenumber();
```

Returns The number of cursors found in the view.

See also: `Cursor(), CursorDelete(), CursorLabel(), CursorLabelPos(), CursorNew(), CursorSet(), HCursorRenumber()`

CursorSet()

This command deletes any existing vertical cursors, then positions a specified number of new cursors, equally spaced in the view and numbered in order from left to right. If any positions are given, they are applied. The cursor labelling style is not changed.

```
Proc CursorSet(num% {,where1 {,where2...}});
```

`num%` The number of cursors in the range 0 to the maximum allowed. 0 turns off all the cursors. It is a run-time error to ask for more than the maximum or less than 0 cursors.

`where` Optional cursor positions in x axis units. Positions that are out of range are set to the nearest valid position.

Examples:

```
CursorSet(0);       'Delete all cursors
CursorSet(2,20,30); 'remove cursors, set 2 at 20 and 30 on x axis.
```

See also: `BinToX(), Cursor(), CursorDelete(), CursorLabel(), CursorLabelPos(), CursorNew(), CursorRenumber()`

Date$()

This function creates a string containing the date. If no arguments are supplied, a string is returned showing the day, month and year in a format specified by the operating system settings. To obtain the date as numbers, use the `TimeDate()` function. For the purpose of this description, we assume that today's date is Wednesday 4 August 2004, the system language is English and the system date separator is "`/`". **Warning**: This command does not exist in version 1.00.

```
Func Date$({dayF%, {monF%, {yearF%, {order%, {sep$}}}}});
```

`dayF%` This sets the format of the day field in the date. This can be written as a day of the week or the day number in the month, or both. If this argument is omitted, the value 2 is used. The options are:

- 1 Show day of week: "`Wednesday`".
- 2 Show the number of the day in the month with leading zeros: "`04`".
- 4 Show the day without leading zeros: "`4`".
- 8 Show abbreviated day of week: "`Wed`".
- 16 Show week day name first, regardless of the `order%` field.

Use 0 for no day field. Add the numbers for multiple options. For example, to return "`Wed 04`", we would enter `11` (1+2+8) as the `dayF%` argument.

If you add 4, 2 is ignored. If you add 8 or 16, 1 is added automatically. If you request both the week day name and the number of the day, the name appears before the number.

`monF%` The format of the month field. This can be returned as either a name or a number. If this argument is omitted, the value 3 is used. The options are:

- 0 No month field.
- 1 Show name of the month: "`August`".
- 2 Show number of month: "`08`".
- 3 Show an abbreviated name of month: "`Aug`".
- 4 Show number of month with no leading zeros: "`8`".

`yearF%` The format of the year field. This can be returned as a two or four digit year.

- 0 No year is shown
- 1 Year is shown in two digits: "98".
- 2 Year is shown in two digits with an apostrophe before it: "'98".
- 3 Year is shown in four digits: "1998".

`order%` The order that the day, month and year appear in the string. If this argument is omitted, the value 0 is used.

- 0 Operating system settings
- 1 month/day/year
- 2 day/month/year
- 3 year/month/day

`sep$` This string appears between the day, month and year fields as a separator. If this string is empty or omitted, Signal supplies a separator based on the system settings.

For example, `Date$(20, 1, 2, 1, " ")` returns `"Wednesday April 1 '98"`. As 20 is 16+4, we have the day first, even through the `order%` argument places the day in between the month and the year. `Date$()` returns `"01/Apr/98"`.

See also: `Seconds(), TimeDate(), Time$()`

Debug()

This function can be used to open the debug window so you can step through your script, set breakpoints and display and edit variables. It can also be used to control access to the debugger by the user.

```
Proc Debug({msg$|opt%});
```

`msg$` When used with no arguments, or with a string argument, this function stops the script as though the Esc key had been pressed or a break-point reached. The debug toobar is shown if it was hidden and, if present, the `msg$` string is displayed in the debug window.

`opt%` When used with an integer argument, the function controls the ability of the user to break into the debugger by pressing Esc. Set `opt%` to 0 to disable Esc, 1 to enable it. Each time a script starts the Esc key is enabled, so you should use this as the first instruction in your script if you want to be certain the user cannot break out.

The `opt%` form of the command was included for use in situations such as student use, where it is important that the user cannot break out of a script by accident. It is advisable to test your script carefully and save it to disk before disabling user breaks; once disabled you cannot stop a looping script except by forcing a fatal error.

See also: `Eval()`

DeleteFrame()

This function deletes the current frame from the current data view. Frames can only be deleted if they were appended and have not yet been saved to disk. It is not possible to delete the last frame in a memory view.

```
Func DeleteFrame();
```

Returns Zero or a negative error code.

See also: `AppendFrame()`, `FrameCount()`, `FrameFlag()`, `FrameTag()`

DelStr$()

This function removes a substring from a string.

```
Func DelStr$(text$, index%, count%);
```

`text$` The string to remove characters from. This string is not changed.

`index%` The start point for the deletion. The first character is index 1. If this is greater than the length of the string, no characters are deleted.

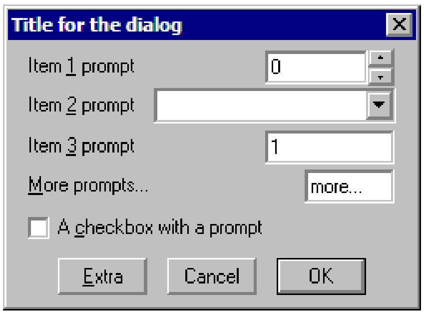
`count%` The number of characters to delete. If this would extend beyond the end of the string, the remainder of the string is removed.

Returns `DelStr$()` returns the original string with the indicated section deleted.

See also: `InStr()`, `LCase$()`, `Len()`, `Mid$()`, `Right$()`, `UCase$()`

Dialogs

You can define your own dialogs to get information from the user. You can define dialogs in a simple way, where each item of information has a prompt and the dialog is laid out automatically, or you can build a dialog by specifying the position of every item. A simple dialog has the structure shown in the diagram:



The dialog is arranged in terms of items. Unless you specifically request otherwise, the dialog items are stacked vertically above each other with buttons arranged at the bottom.

The dialog has a title of your choosing at the top. There are OK and Cancel buttons at the bottom of the dialog. When the dialog is used, pressing the Enter key is equivalent to clicking on OK.

This form of dialog is very easy to program. There is no need to specify any position or size information, the system works it all out. Some users require more complicated dialogs, with control over the field positions. This is also possible, but more involved to program. You are allowed up to 1000 fields in a dialog.

In more complex cases, you specify the position (and usually the width) of the box used for user input. This allows you to arrange data items in the dialog in any way you choose. It requires more work as you must calculate the positions of all the items.

Version 3 extensions

There are new script functions to add and manipulate buttons, to collect a time or x axis value (including using cursor values) and to add a group box. You can also define script functions that are called in response to button presses and user changes to the dialog and an idle-time function that is called repeatedly whilst the dialog is waiting for user actions. All these functions can enable and disable, hide and show and modify dialog items. You can now use `DlgChan()` to get a channel from an XY view. Finally, there are extensions to the integer, real and string fields that allow you to define a drop down list of selectable items to copy into the fields and you can add a spin control to both integer and real numeric fields. All dialogs created in previous versions of Signal should work without any change.

Dialog units

Positions within a dialog are set in *dialog units*. In the x (horizontal) direction, these are in multiples of the maximum width of the characters ‘0’ to ‘9’. In the y (vertical) direction, these are in multiples of the line spacing used for simple dialogs. Unless you intend to produce complex dialogs with user-defined positions, you need not be concerned with dialog units at all.

Simple dialog example

The simple example dialog shown above can be created by this code:

```
var ok%, item1%, item2%, item3, item4$:= "more...", item5;
DlgCreate ("Title for the dialog");          'start new dialog
DlgInteger(1,"Item &1 prompt",0,10,0,0,1); 'range 0-10,spinner
DlgChan   (2, "Item &2 prompt", 1);          'Waveform channel list
DlgReal   (3, "Item &3 prompt", 1.0, 5.0); 'real, range 1.0-5.0
DlgString (4, "&More prompts...", 6);        'string, any characters
DlgCheck  (5, "A &checkbox with a prompt");'a checkbox item
DlgButton (2, "&Extra");                     'extra button, number 2

ok% := DlgShow(item1%, item2%, item3, item4$, item5); 'show dialog
```

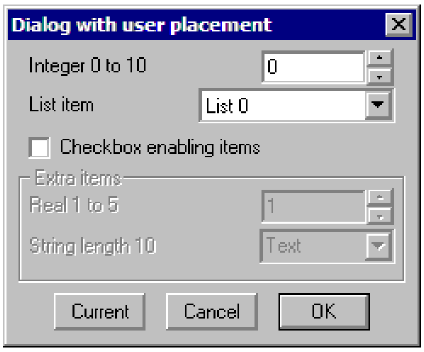
Prompts, & and tooltips

In the functions that set an item with a prompt, if you precede a character in the prompt with an ampersand (&), the following character is underlined and is used by Windows as a short-cut key to move to the field or activate the button. All static dialog items except a group box allow you define a tooltip by appending a vertical bar followed by the tooltip text to the `text$` argument. For example:

```
DlgReal(3, "Rate|Enter the sample rate in Hz", 100, 500);
```

Buttons allow you to specify an additional activation key and an optional tooltip by adding a vertical bar followed by the key code and then another vertical bar followed by the tooltip. See the `label$` argument of `ToolbarSet()` for details of key codes.

More complex example



This example shows how to respond to user actions within a dialog. In this case we use a checkbox to enable and disable a group of items and a button that displays the current values of dialog items. The numbered fields are:

1 An integer, range 0-10 with a spinner
2 A drop list of 4 items
3 A checkbox, used to enable items 4 and 5
4 A real number with a spinner
5 A string with a drop down list of items

We have added button 2 (buttons 0 and 1 are Cancel and OK) and a group box around items 4 and 5. To make room for the group box, the y positions of items 4 and 5 are set explicitly.

With `DlgAllow()` we have set `Func Change%(item%)` to be called whenever the user changes a selection or checkbox or when an editable field loses the input focus. The `item%` argument is set to the item number that changed or to 0 if the dialog is appearing for the first time. We are interested in item 3, the checkbox, and we use the state to enable or disable the group box and the items inside it.

`Func Current%()` is linked to the `"Current"` button. In this case it is used to display a message box that lists the current values of items in the dialog.

```
var ok%, item1%, item2%, item3%, item4, item5$:= "Text", gp%;
DlgCreate("Dialog with user placement",0,0,40,7.5);
DlgInteger(1,"Integer 0 to 10",0,10,0,1,1);      'Int with spinner
DlgList(2,"List item","List 0|List 1|List 2|List 3", 4, 0, 2);
DlgCheck(3, "Checkbox enabling items",0,3);      'checkbox item
DlgReal(4, "Real 1 to 5",1.0,5.0,0,4.5,0.5);     'Real with spinner
DlgString(5, "String length 10",10,"",0,5.5,     'String item with
             "String 1|String 2|String 3");      'drop down list
DlgButton(2, "Current", Current%);               'button+function
DlgAllow(0x3ff, 0, Change%);  'Allow all, no idle, change function
gp% := DlgGroup("Extra items",1,3.8,-1,2.9);     'Group box
ok% := DlgShow(item1%,item2%,item3%,item4,item5$);
Halt;

Func Change%(item%)
var v%;
docase
  case ((item% = 3) or (item% = 0)) then    '0 is initial setup
    v% := DlgValue(3);          'get checkbox state
    DlgEnable(v%, gp%, 4, 5); 'enable groupbox+items 4, 5
endcase;
return 1;                     'Return 1 to keep dialog running
end;
```

```
Func Current%()
var v1%, v2%, v3%, v4, v5$;
v1% := DlgValue(1);          'Retrieve the current values
v2% := DlgValue(2); v3% := DlgValue(3);
v4  := DlgValue(4); v5$ := DlgValue$(5);
Message("Values are %d, %d, %d, %g and %s",v1%,v2%,v3%,v4,v5$);
return 1;                    'Return 1 to keep the dialog running
end;
```

See also: `DlgAllow(), DlgButton(), DlgChan(), DlgCheck(), DlgCreate(), DlgEnable(), DlgGroup(), DlgInteger(), DlgLabel(), DlgList(), DlgReal(), DlgShow(), DlgString(), DlgText(), DlgValue(), DlgVisible(), DlgXValue()`

DlgAllow()

Call this function after `DlgCreate()` and before `DlgShow()` to enable dialog idle time processing, advanced call-back features and dynamic access to the dialog fields. There are no restrictions on what call-back functions can do. However, it is not sensible to place time-consuming code in an idle call-back function or to do anything other than check dialog fields and possible display a warning message in a dialog-item-change function. Call-back functions use `DlgValue()`, `DlgEnable()` and `DlgVisible()` to manipulate the dialog fields. `DlgAllow()` is new in version 3.

```
Proc DlgAllow(allow%{, func id%(){, func ch%()}});
```

`allow%` A number that specifies the actions that the user can and cannot take while interacting with Signal. See `Interact()` for a full description.

`id%()` This is an integer function with no arguments. Use the name with no brackets, for example `DlgAllow(0,Idle%);` where `Func Idle%()` is a script function. When `DlgShow()` executes, the function is called repeatedly in system idle time, as for the `ToolbarSet()` idle function.

If the function return value is greater than 0, the dialog remains open. A zero or negative return value closes the dialog and `DlgShow()` returns the same value.

If this argument is omitted or 0, there is no idle time function.

`ch%()` This is an integer function with one integer argument, for example `Func Changed%(item%)`. You would use `DlgAllow(0,0,Changed%);` to link this function to a dialog. Each time the user changes a dialog item, Signal calls the function with the argument set to the changed item number. There is an initial call with the argument set to 0 when the dialog is about to be displayed.

A field is deemed to change when the user clicks a checkbox or changes a selection in a list or moves the focus from an editable item after changing the text (for real and integer values, the new value must be in range.)

If the change function returns greater than 0, the change is accepted. If the return value is zero, the change is resisted and the focus set back to the changed item. If the return value is negative, the dialog closes and `DlgShow()` returns this value and the arguments are not updated.

See also: `DlgButton(), DlgCreate(), DlgEnable(), DlgShow(), DlgValue(), DlgVisible(), Interact(), ToolbarSet()`

DlgButton()

By default, dialogs created with `DlgCreate()` have a Cancel button (number 0) and an OK button (number 1). You can use this function to add more buttons and to change the text of the existing buttons, or to remove the Cancel button. You can link a script function to a button and use the function return value to decide if the dialog should close. Use this function after `DlgCreate()` and before `DlgShow()`. `DlgButton()` is new in version 3.

```
Proc DlgButton(but%, text${, func ff%{, x, y}});
```

`but%` The button number from 0 to 200. Button 0 is the cancel button, 1 is the OK button. Button numbers higher than 1 create new buttons.

`text$` This sets the button label. Set an empty string to delete a button. You cannot delete button 1; the label is set back to OK if you try. The label text can be followed by an optional key code and an optional tooltip separated by vertical bars. See the `label$` argument of `ToolbarSet()` for details of the format.

If you set a key code, the button can be activated even when the dialog does not have the input focus as long as it is the topmost user dialog and you have not created a toolbar or interact bar from a function linked to the dialog. This allows you to drag cursors in a window, then use the key code without the need to click in the dialog to activate it first.

`ff%()` This is an integer function with no arguments that is called when the button is used. Set the argument to zero or omit it if you don't want a button function, in which case clicking the button closes the dialog, `DlgShow()` returns the button number and the `DlgShow()` arguments are updated for all buttons except 0.

If you supply a function, it is called each time the button is used and the function return value determines what happens next:

<0 The button acts as the Cancel button. The dialog closes and `DlgShow()` returns this value and its arguments are not updated.

0 The button acts as OK. The dialog closes and the `DlgShow()` return value is the button number and its arguments are updated.

>0 The dialog continues to display.

The button function can use `DlgEnable()`, `DlgValue()` and `DlgVisible()`.

`x,y` Set the button position, both or neither of these must be supplied. If the button position is not supplied it will be positioned at the bottom of the dialog.

See also: `DlgAllow()`, `DlgCreate()`, `DlgEnable()`, `DlgShow()`, `DlgValue()`, `DlgVisible()`, `ToolBarSet()`

DlgChan()

You often need to select a channel of a particular type from a view. This function defines a dialog entry that lists channels that meet a specification. For simple dialogs, the `wide`, `x` and `y` arguments are not used. Channel lists are checked or created when the `DlgShow()` function runs. If the current view is not a data or XY view, the list will be empty.

```
Proc DlgChan(item%, text$|wide, mask%|list%[]{, x{, y}});
```

`item%` This sets the item number in the dialog.

`text$` The text to display as a prompt. If the prompt contains a vertical bar, "|" any following text will be used as a tooltip and displayed when the mouse pointer is held over the item.

`wide` This is an alternative to the prompt. It sets the width of the box in which the user selects a channel. If the width is not given the number entry box has a default width of the longest channel name in the list or 12, whichever is the smaller.

`mask%` This is an integer code that determines the channels to be displayed. It is ignored for XY views. You can select channels of particular types by adding together the following codes:

1 Waveform channels
2 Marker channels
4 Idealised traces

If none of the above values are used, then the list includes all channels. The following codes can be added to exclude channels from the list created above:

1024 Exclude visible channels
2048 Exclude hidden channels
4096 Exclude selected channels
8192 Exclude non-selected channels

Finally, adding the following codes allows special entries to be added to the list:

131072 Add `None` as an entry in the list (returns 0)
262144 Add `All channels` as an entry in the list (returns -1)
524288 Add `All visible channels` as an entry in the list (returns -2)
1048576 Add `Selected` as an entry in the list (returns -3)

`list%` As an alternative to a mask, you can pass in a channel list (as constructed by `ChanList()`). This must be an array of channels, with the first element of the array holding the number of channels in the list.

`x` If omitted or zero, the selection box is right justified in the dialog box, otherwise this sets the position of the left end of the channel selection box.

`y` If omitted, this takes the value of `item%`. It is the position of the bottom of the channel selection box.

The variable passed to `DlgShow()` for this field should be an integer. If the variable passed in holds a channel number in the list, the field shows that channel, otherwise it shows the first channel in the list (usually `None`). The result from this field in `DlgShow()` is a channel number, or 0 if `None` is selected, -1 if `All channels` is selected, -2 if `All visible channels` is selected or -3 if `Selected` is chosen.

See also: `DlgAllow(), DlgButton(), DlgCheck(), DlgCreate(), DlgInteger(), DlgLabel(), DlgList(), DlgReal(), DlgShow(), DlgString(), DlgText()`

DlgCheck()

This defines a dialog item that is a check box (on the left) with a text string to its right. For simple dialogs, the `x` and `y` arguments are not used.

```
Proc DlgCheck(item%, text${, x{, y}});
```

`item%` This sets the item number in the dialog.

`text$` The text to display to the right of the check box. If the prompt contains a vertical bar, "|" any following text will be used as a tooltip and displayed when the mouse pointer is held over the item.

`x,y` The position of the bottom left hand corner of the check box. If omitted, x is set to 1 and y to `item%`. When used without these fields, this function behaves exactly like the simple dialog functions, and can be mixed with them.

When `DlgShow()` is used, the box is checked if the variable passed in is non-zero and unchecked if it is zero. The variable passed should be an integer and is returned as 0 for unchecked or 1 for checked.

See also: `DlgAllow(), DlgButton(), DlgChan(),DlgCreate(), DlgInteger(), DlgLabel(), DlgList(), DlgReal(), DlgShow(), DlgString(), DlgText()`

DlgCreate()

This function starts the definition of a dialog. It also kills off any previous dialog that might be partially defined. For simple dialogs, the optional arguments are not used.

```
Func DlgCreate(title${,x,y,wide,high,help%|help${,scr%{,rel%}}});
```

`title$` A string holding the title for the dialog.

`x,y` Optional, taken as 0 if omitted. The position of the top left hand corner of the dialog. The positions are in percentages of the screen size. The value 0 means centre the dialog. Values out of the range 0 to 95 are limited to the range 0 to 95.

`wide` The width of the dialog in dialog units. If this is omitted, or set to 0, Signal works out the width for itself, based on the items in the dialog.

`high` The height of the dialog in dialog units. If omitted, or set to 0, Signal works it out for itself, based on the dialog contents.

`help` This is a string or numeric identifier that identifies the help page to be displayed if the user requests help when the dialog is displayed. This argument is ignored if your version of Signal doesn't support help.

`scr%` Optional screen selector. See `Window()` command for details.

`rel%` Omit or set 0 for application window relative, 1 for screen or desktop relative.

Returns This function returns 0 if all was well, or a negative error code.

For simple use, only the first argument is needed. The remainder are for use with more complicated menus where precise control over menu items is required.

Use of & in prompts

In the functions that set an item with a prompt, if you precede a character in the prompt with an ampersand "`&`", the following character is used by Windows as a short-cut key to move to the field and the character is underlined. Ampersand characters are ignored on systems that do not use this mechanism.

See also: `DlgAllow(), DlgButton(), DlgChan(), DlgCheck(), DlgInteger(), DlgLabel(), DlgList(), DlgReal(), DlgShow(), DlgString(), DlgText(), Window()`

DlgEnable()

This can only be used from a dialog call-back function to enable or disable dialog items. There are two versions of this command. With a single argument, it returns the enabled state of an item; with two or more arguments it sets the enabled state of one or more dialog items. When you enable or disable an item, any prompt or spin control associated with the item is also enabled or disabled. This function is new in version 3.

```
Func DlgEnable(en%, item%|item%[]{, item%|item%[]...});
Func DlgEnable(item%);
```

`en%` Set 0 to disable list items, 1 to enable them and 2 to enable and give the first item the input focus. Input focus changes should be used sparingly to avoid user confusion; they can cause button clicks to be missed.

`item%` An item number or an array of item numbers of dialog elements. The item number is either the number you set, or the number returned by `DlgText()` or `DlgGroup()`, or `-button`, where `button` is the button number. You cannot access prompts separately from their items as this makes no sense.

Returns When called with a single argument it returns the enabled state of the item, otherwise it returns 0.

See also: `DlgAllow(), DlgButton(), DlgCreate(), DlgShow(), DlgValue(), DlgVisible()`

DlgGroup()

This routine creates a group box, which is a rectangular frame with a text label at the top left corner. You can use this between calls to `DlgCreate()` and `DlgShow()`. There is nothing for the user to edit in this item, so you do not supply an item number and there is no matching argument in `DlgShow()`. However, the returned number is an item number (above the values used to match items to `DlgShow()` arguments) that you can use in call-back functions to identify the group box. This function was added at version 3.

```
Func DlgGroup(text$, x, y, width, height);
```

`text$` The text to display at the top left of the group box.

`x,y` The position of the top left corner of the group box.

`width` If positive, the width of the group box. If negative, this is the offset of the right hand side of the group box from the right hand edge of the dialog.

`height` The height of the group box.

Returns The routine returns an item number so that you can refer to this in call-back functions to use `DlgVisible()` and `DlgEnable()`.

See also: `DlgCreate(), DlgEnable(), DlgShow(), DlgVisible()`

DlgInteger()

This function defines a dialog entry that edits an integer with an optional spin control or drop down list of selectable items. The numbers you enter may not contain a decimal point. For simple dialogs, the `wide`, `x`, `y`, `sp%` and `li$` arguments are not used.

```
Proc DlgInteger(item%, text$|wide, lo%, hi%{, x{, y{, sp%|li$}}});
```

`item%` This sets the item number in the dialog in the range 1 to the number of items.

`text$` The prompt to display, optionally followed by a vertical bar and tooltip text.

`wide` This is an alternative to the prompt. It sets the width in dialog units of the box in which the user types the integer. If the width is not given the number entry box has a default width of 11 digits (or width needed for the number range?).

`lo%` The start of the range of acceptable numbers.

`hi%` The end of the range of acceptable numbers.

`x` If omitted or zero, the number entry box is right justified in the dialog box, otherwise this sets the position of the left end of the box in dialog units.

`y` If omitted or zero, this takes the value of `item%`. It is the position of the bottom of the number entry box in dialog units.

`sp%` If present and non-zero, this adds a spin box with a click increment of `sp%`.

`li$` If present, this argument is a list of items separated by vertical bars that can be selected into the integer field, for example `"1|10|100"`.

The variable passed into `DlgShow()` should be an integer. The field starts with the value of the variable if it is in the range. Otherwise, it is limited to the nearer end of the range.

See also: `DlgAllow(), DlgButton(), DlgChan(), DlgCheck(), DlgCreate(), DlgLabel(), DlgList(), DlgReal(), DlgShow(), DlgString(), DlgText()`

DlgLabel()

This function sets an item that has no editable part, that is an item used as a label. For simple dialogs, the `wide`, `x` and `y` arguments are not used. You can add text to a dialog without using an item number with `DlgText()`.

```
Proc DlgLabel(item%, text${, x{, y}});
```

`item%` This sets the item number in the dialog.

`text$` The text to display. If the label contains a vertical bar, "|" any following text will be used as a tooltip and displayed when the mouse pointer is held over the item.

`x` If omitted, the text is left justified in the dialog box, otherwise this sets the position of the left end of the text in the dialog.

`y` If omitted, this takes the value of `item%`. It is the position of the bottom of the text in the dialog.

When you call `DlgShow()`, you must provide a dummy variable for this field. The variable is not changed and can be of any type, but must be present.

See also: `DlgAllow()`, `DlgButton()`, `DlgChan()`, `DlgCheck()`, `DlgCreate()`, `DlgInteger()`,`DlgList()`, `DlgReal()`, `DlgShow()`, `DlgString()`, `DlgText()`

DlgList()

This defines a dialog item for a one of n selection. Each of the possible items to select is identified by a string. For simple dialogs, the `wide`, `x` and `y` arguments are not used.

```
Proc DlgList(item%, text$|wide, list$[]|str${, n%{, x{, y}}});
```

`item%` This sets the item number in the dialog.

`text$` The text to display as a prompt. If the prompt contains a vertical bar, "|" any following text will be used as a tooltip and displayed when the mouse pointer is held over the item.

`wide` This is an alternative to the prompt. It sets the width of the box in which the user selects an item. If the width is not given the number entry box has a default width of the longest string in the list or 18, whichever is the smaller.

`list$` An array of strings. These hold the items to be presented in the list. Each string should not be over 18 characters long, or it will be truncated.

`str$` An alternative way to define the items to be presented in the list. The single string holds all of the items, items are separated by the vertical bar character (|). Again, items should not be more than 18 characters long.

`n%` The number of entries in the list. If this is omitted, or if it is larger than the array, then the size of the array is used. If the `str$` form of the command is used, the number of items in the string sets the maximum number.

`x` If omitted or zero, the selection box is right justified in the dialog, otherwise this sets the position of the left end of the selection box.

`y` If omitted, this takes the value of `item%`. It is the position of the bottom of the list selection box.

The result obtained from this is the index into the list of the list element chosen. The first element is number 0. The variable passed to `DlgShow()` for this item should be an integer. If the value of the variable is in the range `0` to `n%-1`, this sets the item to be displayed, otherwise the first item in the list is displayed.

The following example shows how to set a list:

```
var list$,ok%,which%:=0;   'string list, test for OK, result
list$ := "zero|one|two|three";           'these are the choices
DlgCreate("List example");    'Start the dialog
DlgList(1,"Make your choice", list$);
ok% := DlgShow(which%);       'Display dialog, wait for user
```

See also: `DlgAllow(), DlgButton(), DlgChan(), DlgCheck(), DlgCreate(), DlgInteger(), DlgLabel(), DlgReal(), DlgShow(), DlgString(), DlgText()`

DlgReal()

This function defines a dialog entry that edits a real number. For simple dialogs, the `wide`, `x` and `y` arguments are not used.

```
Proc DlgReal(item%, text$|wide, lo, hi{, x{, y{, sp|li$}}});
```

`item%` This sets the item number in the dialog in the range 1 to the number of items.

`text$` The prompt to display, optionally followed by a vertical bar and tooltip text.

`wide` This is an alternative to the prompt. It sets the width in dialog units of the box in which the user types a real number. If `wide` is not given the box has a default width of 12 digits.

`lo,hi` The range of acceptable numbers.

`x` If omitted or zero, the number edit box is right justified in the dialog, otherwise this sets the position in dialog units of the left end of the number entry box.

`y` Bottom of the number edit box position. If omitted, the value of `item%` is used.

`sp` If present and non-zero, this adds a spin box with a click increment of `sp`.

`li$` If present, this argument is a list of items separated by vertical bars that can be selected into the editing field, for example `"1.0|10.0|100.0"`.

The variable passed into `DlgShow()` should be a real number. The field will start with the value of the variable if it is in the range, otherwise the value is limited to `lo` or `hi`.

See also: `DlgChan(), DlgCheck(), DlgCreate(), DlgInteger(), DlgLabel(), DlgList(),DlgShow(), DlgString(), DlgText(), DlgXValue()`

DlgShow()

This function displays the dialog you have built and returns values from the fields identified by item numbers, or makes no changes if the user kills the dialog with the Cancel button. Once the dialog has been dismissed, all information about it is lost. You must create a new dialog before you can use this function again.

```
Func DlgShow(&item1|item1[], &item2|item2[], &item3|item3[] ...);
```

For each item that you have defined, you must provide a variable of a suitable type to receive the result. It is an error to pass the wrong type of variable, except in the case of an integer field which you can return into a real or an integer variable. Items created with `DlgLabel()` must have a variable too, even though it is not changed.

These variables also set the initial values of the fields for editing. If an initial value is out of range or not allowed, the value is changed to the nearest legal value. In the case of a string, illegal characters are deleted before display.

In addition to passing a simple variable, you can pass an array. An array with `n` elements matches `n` items in the dialog. The array type must match the items.

If the user clicks on OK, all the variables are updated to their new values. If the user clicks on Cancel, the variables are not changed.

Returns The function returns 0 if the user clicked on the Cancel button, or 1 if the user clicked on OK.

See also: `DlgAllow(), DlgButton(), DlgChan(), DlgCheck(), DlgCreate(), DlgInteger(), DlgLabel(), DlgList(), DlgReal(), DlgString(), DlgText()`

DlgString()

This defines a dialog entry that edits a text string. You can limit the characters that you will accept in the string. For simple dialogs, the `wide`, `x` and `y` arguments are not used.

```
Proc DlgString(item%, text$|wide, max%{, legal${, x{,y{,sel$}}}});
```

`item%` This sets the item number in the dialog in the range 1 to the number of items.

`text$` The prompt to display, optionally followed by a vertical bar and tooltip text.

`wide` This is an alternative to the prompt. It sets the width in dialog units of the box in which the user types the string. If the width is not given the number entry box has a default width of `max%` or 60, whichever is the smaller.

`max%` The maximum number of characters allowed in the string.

`legal$` A list of acceptable characters. See `Input$()` for a full description. If this is omitted, or an empty string, all characters are allowed.

`x` If omitted or zero, the string entry box is right justified in the dialog, otherwise this sets the position in dialog units of the left end of the string entry box.

`y` If omitted or zero, this takes the value of `item%`. It is the position of the bottom of the string entry box in dialog units.

`sel$` If this string is present, it should hold a list of items separated by vertical bars, for example `"one|two|three"`. The field becomes an editable combo box with the items in the drop down list. This is new at version 5.

The result from this operation is a string of legal characters. The variable passed to `DlgShow()` should be a string. If the initial string set in `DlgShow()` contains illegal characters, they are deleted. If the initial string is too long, it is truncated.

See also: `DlgAllow(), DlgButton(), DlgChan(), DlgCheck(), DlgCreate(), DlgInteger(), DlgLabel(), DlgList(), DlgReal(), DlgShow(), DlgText()`

DlgText()

This places non-editable text in the dialog box. This is different from `DlgLabel()` as you do not supply an item number and it does not require a variable in the `DlgShow()` function. It returns an item number (higher than item numbers for matching arguments in `DlgShow()`) that you can use to identify this field in call-back functions, for example `DlgVisible()`.

```
Func DlgText(text$, x, y{, wide});
```

`text$` The prompt to display, optionally followed by a vertical bar and tooltip text.

`x,y` The position of the bottom left hand corner of the first character in the string, in dialog units. Set `x` to 0 for the default label position (the same as `DlgLabel()`).

`wide` Normally, the width of the field is set based on `text$`. This optional argument sets the width in dialog units. This allows you to replace the text with a longer string from a call-back function.

Returns An item number to identify this field for call-back functions.

See also: `DlgAllow()`, `DlgButton()`, `DlgChan()`, `DlgCheck()`, `DlgCreate()`, `DlgInteger()`, `DlgLabel()`, `DlgList()`, `DlgReal()`, `DlgShow()`, `DlgString()`

DlgValue() and DlgValue$()

These functions can only be used from a dialog call-back function to get and optionally set the value of an item, item prompt or button text. They are new in version 3.

```
Func DlgValue(item%{, val});
Func DlgValue$(item%{, val$});
```

`item%` This identifies the dialog item. For items with arguments in `DlgShow()`, use the `item%` value you set to create the field. For items created with `DlgText()` and `DlgGroup()`, use the returned item number. For buttons use minus the button number. To access the prompt for an item add 1000 to the item number you set.

`val` This optional argument holds the new item value. If you omit this argument the item is not changed. You can use numeric values on numeric fields or to set a checkbox or a list. You can use a text value to change a prompt or button label or to change the text of an editable control or to select the first matching item in a list box. It is up to you to make sure the text is acceptable for editable items.

Returns The returned value is the current value of the item. You can use `DlgValue$()` on any item to get the current contents of the field, checkbox text, button or prompt as a text string. Use `DlgValue()` to collect numeric or checkbox values.

If there is a problem running the command, for example if the item does not exist, or an argument type is not appropriate for an item, the result is an empty string or the value 0.

See also: `DlgAllow()`, `DlgCreate()`, `DlgEnable()`, `DlgShow()`, `DlgVisible()`

DlgVisible()

This can only be used from a dialog call-back function to show or hide dialog items. There are two versions of this command. The version with a single argument returns the visible state of an item; the version with two or more arguments sets the visible state of one or more dialog items. When you show or hide an item, any prompt or spin control associated with the item is also shown or hidden. This function is new in version 3.

```
Func DlgVisible(show%, item%|item%[]{, item%|item%[]...});
Func DlgVisible(item%);
```

`show%` Set this to 1 to show the items in the list and to 0 to hide them.

`item%` An item number of an element of the dialog or an integer array containing a list of item numbers. The item numbers are either the number you set, or the number returned by `DlgText()` or `DlgGroup()`, or `-button`, where `button` is the button number. You cannot access prompts separately from their items as this makes no sense.

Returns When called with a single argument it returns the visible state of the item, otherwise the return value is 0.

See also: `DlgAllow()`, `DlgCreate()`, `DlgEnable()`, `DlgShow()`, `DlgValue()`

DlgXValue()

This creates an editable combo box to collect an x axis value for the current file, memory or XY view. The combo box drop down list is populated with cursor positions and other window values when `DlgShow()` runs. If the current view is not suitable, the list is empty. This control accepts expressions, for example: `(Cursor(1)+Cursor(2))/2`. The matching `DlgShow()` argument is a real number to hold a time in seconds for a file view, or an x axis value for other views. This procedure is new in version 3.

```
Proc DlgXValue(item%, text$|wide{, x{, y}});
```

`item%` This sets the item number in the dialog in the range 1 to the number of items.

`text$` The text to display as a prompt. If the prompt contains a vertical bar, "|" any following text will be used as a tooltip and displayed when the mouse pointer is held over the item.

`wide` This is an alternative to the prompt. It sets the width in dialog units of the combo box. If the width is not given the combo box has a default width of 18 numbers.

`x` If omitted or zero, the string entry box is right justified in the dialog, otherwise this sets the position in dialog units of the left end of the string entry box.

`y` If omitted or zero, this takes the value of `item%`. It is the position of the bottom of the string entry box in dialog units.

See also: `DlgAllow()`, `DlgButton()`, `DlgChan()`, `DlgCheck()`, `DlgCreate()`, `DlgInteger()`, `DlgLabel()`, `DlgList()`, `DlgReal()`, `DlgShow()`, `DlgString()`, `DlgText()`

Draw()

This allows invalid regions in the current view to update. `Draw()` on a view that is up-to-date should make no change. The view is not brought to the front.

```
Proc Draw({from {, size}});
```

`from` The left hand edge of the view in x axis units.

`size` The width of the view in x axis units.

With no arguments, `Draw()` updates invalid areas in the view. With one argument, the view is scrolled to start at `from`. With two arguments, the width is set (unless it is unchanged) and then it is drawn. If `size` is negative or omitted, the same size as last time is used.

Data views run from `Mintime()` to `Maxtime()`. There are no limits set on the range of values for an XY view axis. However, huge numbers cause ugly axis labels.

Beware! When `Draw()` makes a view scroll, it scrolls by an integral number of pixels (otherwise the image would be disjointed). Thus, the requested start may not be the same as that reported by `XLow()`. The requested value will be in the first pixel. With an x axis in seconds the following code may not move the display at all if a pixel is more than a second wide:

```
Draw(XLow()+1.0); 'This usually steps by just under 1 second
```

See also: `DrawAll(), XRange(), XLow(), XHigh(), Maxtime(), Mintime()`

DrawAll()

This routine updates all views with invalid regions. This is equivalent to iterating through all the views and performing a `Draw()` on each.

```
Proc DrawAll();
```

See also: `Draw()`

DrawMode()

This sets and reads the display mode for the channel in a data view. You can set the display mode for channels that are not displayed.

```
Func DrawMode(cSpc{, mode%{, dotSz%|binSz|offset{,
                        err%|flags%}}});
```

`cSpc` A channel list specifier of the channels to use. See the *Script language syntax* chapter for a definition of channel specifiers.

`mode%` If present and positive, this sets the display mode, and returns the previous mode. If an inappropriate mode is requested, no change is made. Some modes require additional parameters (for example a dot size). If additional parameters are omitted, the last known values are used. The mode values for setting draw modes in the view are:

0	The standard mode for the channel.
1	Dots for markers, using the `dotSz%` argument if it is provided.
2	Lines for markers.
3	Rate for markers.
4	Histogram for waveform.
5	Line for waveform.
6	Dots for waveform, using the `dotSz%` argument if it is provided.
7	Skyline for waveform.
8	Cubic spline for waveform.
9 to 14	Reserved.
15	Basic for idealised trace.
16	Convolution for idealised trace.
17	Both basic and convolution of idealised trace.

If present and negative, the function returns stored bin size or dot size. The mode values for getting data are:

-1	Reserved, does nothing and returns -1
-2	Return the current dot size
-3	Return the current bin size for a histogram
-4	Return the offset of the basic idealised trace
-5	Return the draw mode flags
-13	Return the current drawing style for errors

Values in the range -4 to -12 are reserved for future use. If `mode%` is absent no changes are made.

`dotSz%` This sets the dot size to use on screen or the point size to use on a printer. 0 is the smallest size available. The maximum size allowed is 10. Set to -1 for no change.

`binSz` This sets the width of the rate histogram bins.

`Offset` The y-offset of the idealised trace from the raw data.

`err%` The drawing style for errors: 0=none, 1=1 SEM, 2=2 SEM, 3=SD.

`flags%` Set to 1 to show baseline otherwise 0.

Returns The draw mode before the call if a single channel is set or a value determined by a negative `mode%` value. For multiple channels or an invalid call, it returns -1.

See also: `Draw(), ViewStandard(), XYDrawMode(), ChanValue()`

Dup()

This gets the view handle of a duplicate of the current view, or the number of duplicates, including the original. Duplicated views are numbered from 1 (1 is the original). If a duplicate is deleted, higher numbered duplicates are renumbered. For more information on this see `WindowDuplicate()`.

```
Func Dup({num%});
```

`num%` The number of the duplicate view to find, starting at 1. You can also pass 0 (or omit `num%`) as an argument, to return the number of duplicates.

Returns If `num%` is greater than 0, this returns the view handle of the duplicate, or 0 if the duplicate does not exist. If `num%` is 0 or omitted, this returns the number of duplicates including the original. The following illustrates the use of `Dup()`.

```
var maxDup%, i%, dvh%;      'declare variables
maxDup% := Dup(0);          'Get maximum numbered duplicate
for i% := 1 to maxDup% do  'loop round all possible duplicates
  dvh% := Dup(i%);          'get handle of this duplicate
  if (dvh% > 0) then        'does this duplicate exist?
   PrintLog(view(dvh%).WindowTitle$()+"\n");'print window title
  endif;
next;
```

See also: `App()`, `View()`, `WindowDuplicate()`

EditClear()

In a memory view, this command zeros the data in all channels. In a text view, it deletes any selected text.

```
Func EditClear();
```

Returns The function returns 0 if nothing was deleted, otherwise it returns the number of items deleted or 1 if the number is not known, or a negative error code.

See also: `EditCopy()`, `EditCut()`

EditCopy()

This command copies data from the current view to the clipboard. The effect depends on the type of the current view. Text views copy data as text. Data views copy as a bitmap, as a scaleable image, as text in the format set by `ExportTextFormat()`, `ExportChanFormat()` and `ExportChanList()` or as binary data in a private format.

```
Func EditCopy({as%});
```

`as%` This sets how to copy data when several formats are possible. If omitted, all formats are used. When only one format is possible, the argument is ignored. The format value is the sum of:

1 Copy as a bitmap
2 Copy as a scaleable image (Windows metafile)
4 Copy as text
8 Copy as binary data in the CED private format

Returns It returns the sum of the format specifiers for each format exported from a data view, or the number of characters exported from a text view. It returns 0 if nothing was placed on the clipboard.

See also: `EditSelectAll()`, `ExportTextFormat()`, `ExportChanFormat()`, `ExportChanList()`, `EditClear()`, `EditCut()`, `EditPaste()`

EditCut()

This command cuts data from the current text or script view to the clipboard.

```
Func EditCut({as%});
```

`as%` This optional argument sets how data is copied to the clipboard in cases where there are several formats possible. Currently it is ignored.

Returns Returns the number of characters placed on the clipboard.

See also: `EditClear(), EditCopy(), EditPaste()`

EditPaste()

You may only paste into a text view when the clipboard contains text. The contents of the clipboard are inserted at the current caret. If text is already selected, it is replaced by the clipboard contents. If the clipboard contains binary data in the CED private format, this data can be pasted into a data view.

```
Func EditPaste();
```

Returns The function returns the number of characters inserted.

See also: `EditCopy(), EditCut()`

EditSelectAll()

This function selects all items in the current text view that can be copied to the clipboard. This is the same as the Edit menu Select All option.

```
Func EditSelectAll();
```

Returns It returns the number of selected items that could be copied to the clipboard.

See also: `EditCopy(), EditClear(), EditCut()`

Error$()

This function converts a negative error code returned by a function into a text string.

```
Func Error$(code%);
```

`code%` A negative error code returned from a Signal function.

Returns It returns a string that describes the error.

Eval()

This evaluates the argument and converts the result into text. The text is displayed in the Script view or the Evaluate window message area, as appropriate, when the script ends. The argument can be the value returned by a function.

```
Proc Eval(arg);
```

`arg` A real or integer number or a string.

If you use `Eval()` it will suppress any run-time error messages as it uses the same mechanism as the error system. A common use of `Eval()` in a script is to report an error condition during debugging, for example:

```
if val<0 then Eval("Negative value"); Halt; endif;
```

Another use of `Eval()` is in the Script menu Evaluate window to see the result returned by a function or expression, as in these examples:

```
Eval(FileDelete(myfile$)); ' display 1 or a negative error code
Eval(Error$(-1531));       ' give string for error code if known
```

See also: `Debug()`, `Error$()`, `Print()`, `PrintLog()`

Exp()

This function calculates the exponential function (e to the power of x) for a single value, or replaces the elements of a real array by their exponentials. If a value is too large, overflow will occur, causing the script to stop for single values, and a negative error code for arrays.

```
Func Exp(x|x[]{[]...});
```

`x` The argument for the exponential function or an array of real values.

Returns With an array, the function returns 0 if all was well, or a negative error code if a problem was found (such as overflow).

With an expression, it returns the exponential of the number.

ExportChanFormat()

This command sets the channel text export format for use by `FileExportAs()` and `EditCopy()`. It is equivalent to the data, time and headings settings for each channel type in the Text Output Configuration dialog. Using `ExportTextFormat()` with no arguments will reset these fields to enable the output of data, time and headings for the waveform channel type only.

```
Proc ExportChanFormat(type%, data%, xval%, heads%);
```

`type%` The type of data to set the format for:

0	Waveform	1	Marker	4	Fitted	5	Idealised

`data%` Set this non-zero to enable data output for this channel type.

`xval%` Set this non-zero to enable output of x axis values for this channel type. For a waveform channel this is ignored when `data%` is disabled.

`heads%` Set this non-zero to enable output of column headings for this channel type. This is ignored if neither `data%` nor `xval%` are enabled.

See also: `EditCopy()`, `FileExportAs()`, `ExportChanList()`, `ExportFrameList()`, `ExportTextFormat()`, `ExportTimeRange()`

ExportChanList()

This command sets a channel list to export for use by `FileExportAs()` and `EditCopy()` from a data view.

```
Proc ExportChanList(cSpc);
```

`cSpc` A channel list specifier of the channels to export. See the *Script language syntax* chapter for a definition of channel specifiers.

See also: `EditCopy()`, `FileExportAs()`, `ExportChanFormat()`, `ExportFrameList()`, `ExportTextFormat()`, `ExportTimeRange()`

ExportFrameList()

This command sets a list of frames for use by `FileExportAs()` and `EditCopy()`.

```
Proc ExportFrameList(sFrm%{, eFrm%{,mode%}});
Proc ExportFrameList(frm$|frm%[]{,mode%});
```

`sFrm%` First frame to export. This option processes a range of frames. `sFrm%` can also be a negative code as follows:

-1 All frames in the file are included
-2 The current frame
-3 Frames must be tagged
-6 Frames must be untagged

`eFrm%` Last frame to export. If this is -1 the last frame is the last in the data view. This argument is ignored if `sFrm%` is a negative code.

`frm$` This option specifies a list of frames using a string such as "1..32,40,50".

`frm%[]` An array of frame numbers to process. This option provides a list of frame numbers. The first element holds the number of frames in the list.

`mode%` If `mode%` is present it is used to supply an additional criterion for including each frame in the range, list or specification. If `mode%` is absent all frames are included. The modes are:

0-n Frames must have a state matching the value of `mode%`
-1 All frames in the specification are processed
-2 Only the current frame, if it is in the main list, is processed
-3 Frames must also be tagged
-6 Frames must also be untagged

The following simple example exports all frames to `fred.cfs`.

```
ExportFrameList(-1);           'export from all frames in the view
FileExportAs ("fred.cfs", 1);  'export selected data as text
```

See also: `EditCopy()`, `FileExportAs()`, `ExportChanList()`, `ExportChanFormat()`, `ExportTextFormat()`, `ExportTimeRange()`

ExportTextFormat()

This command sets the text export format for use by `FileExportAs()` and `EditCopy()`. It is equivalent to setting decimal places, field width, string delimiter, item separator and frame header in the Text Output Configuration dialog. The command with no arguments resets everything in the dialog to default settings: decimal places to 5, field width to 0, the string delimiter to double quotes, the separator to a tab character, the header disabled. It also enables the output of data, time and headings for the waveform channel type only. See `ExportChanFormat()` to set these.

```
Proc ExportTextFormat({dDec%, tDec%, width%, lim$, sep${, head%{,
                        intp%}}});
```

`dDec%` Decimal places for data values.

`tDec%` Decimal places for time values.

`width%` Field width for all values, or zero for minimum width.

`lim$` The delimiter, which is the character to place at the start and end of each text string in the output. The normal character to use is a double-quote mark.

`sep$` The separator character which is used to separate multiple data items on a line. This should be one of tab, comma or space.

`head%` If this is present and non-zero, Signal will output the frame header information.

`intp%` The interpolation method to use for waveform output. The default value is 0.

0 None 1 Linear 2 Cubic Spline.

See also: `EditCopy()`, `FileExportAs()`, `ExportChanList()`, `ExportFrameList()`, `ExportChanFormat()`, `ExportTimeRange()`

ExportTimeRange()

This command sets an x axis range for use by `FileExportAs()` and `EditCopy()` in a data view. This is equivalent to setting start and end times in the export setup dialogs.

```
Proc ExportTimeRange(sRange, eRange);
```

`sRange` The start of the range of data to export, in x axis units.

`eRange` The end of the range of data to export, in x axis units.

See also: `EditCopy(), FileExportAs(), ExportChanFormat(), ExportChanList(), ExportFrameList(), ExportTextFormat()`

FileClose()

This is used to close the current window or external file. You can supply an argument to close all views associated with the current data view or to close all the views belonging to the application.

```
Func FileClose({all% {,query%});
```

`all%` This argument determines the scope of the file closing. Possible values are:

- -1 Close all views except loaded scripts and debug windows
- 0 Close the current view. This is the same as omitting `all%`
- 1 Close all windows associated with the current view

`query%` This determines what happens if a view holds unsaved data:

- -1 Don't save the data or query the user
- 0 Query the user about each view that needs saving. If the user chooses Cancel, the operation stops, leaving all unclosed windows behind. This is the same as omitting `query%`.

Returns The number of views that have not been closed. This can occur if a view needs saving and the user requests Cancel.

Note: A common fault in scripts is the use of the construct:

```
View(v%).FileClose(0);
```

This can cause problems because, if the current view is already `View(v%)`, then at the end of the function the script will attempt to switch back to `View(v%)` again, but it is now gone! This results in a "`View is wrong type`" error for no obvious reason. To avoid the problem use:

```
View(v%);
FileClose(0);
```

See also: `FileOpen(), FileSave(), FileSaveAs(), FileNew()`

FileComment$()

This function accesses the file comment in the file associated with the current file or memory view. File comments for XY and text based views are always blank. The comment string is up to 72 characters in length.

```
Func FileComment$({,new$});
```

`new$` If present, the command replaces the existing comment with `new$`.

Returns The comment at the time of the call.

See also: `FrameComment$()`

FileConvert$()

This function converts a data file from a “foreign” format into a Signal data file. The range of foreign formats supported depends on the number of import filters in the `Signal3\import` folder.

```
Func FileConvert$(src${, dest${, flag%{ ,&err%}}});
```

`src$` This is the name of the file to convert. The file extension is used to determine the file type (unless `flag%` bit 0 is set). Known file extensions include: `abf`, `cfs`, `cnt`, `cut`, `dat`, `eeg`, `ewb`, `ibw`, `son` and `uff`. We expect to add more. If an empty string is used or one containing wild cards then a file selection dialog will appear.

`dest$` If this is present, it sets the destination file. If this is not a full path name, the name is relative to the current directory. If you do not supply a file extension then Signal appends `".cfs"`. If you set any other file extension, Signal cannot open the file as a Signal data file. If you do not supply this argument, the converted file will be written to the same folder as the source file, using the original file name with the file extension changed to `.cfs`.

`flag%` This argument is the sum of the flag values: 1=Ignore the file extension of the source file and try all possible file converters, 2=Allow user interaction if required (otherwise sensible, non-destructive defaults are used for all decisions).

`err%` Optional integer variable that is returned as 0 if the file was converted, otherwise it is returned holding a negative error code.

Returns The full path name of the created file, or an empty string if the file was not converted.

See also: `FileOpen(), FilePath$(), FilePathSet(), FileList()`

FileCopy()

This function copies a source file to a destination file. File names can be specified in full or relative to the current directory. Wildcards cannot be used.

```
Func FileCopy(src$, dest${, over%});
```

`src$` The source file to copy to the destination. This file is not changed.

`dest$` The destination file. If this file exists you must set `over%` to overwrite it.

`over%` If this optional argument is 0 or omitted, the copy will not overwrite an existing destination file. Set to 1 to overwrite.

Returns The routine returns 1 if the file was copied, 0 if it was not. Reasons for failure include: no source file, no destination path, insufficient disk space, destination exists and insufficient rights.

See also: `BRead(), BWrite(), FileDelete(), FileOpen(), ProgRun()`

FileDelete()

This function deletes one or more files. File names can be specified in full, or relative to the current directory.

Windows file names are of the form `x:\folder1\folder2\foldern\file.ext` or `\\machine\folder1\folder2\foldern\file.ext` across a network. If a name does not start with a `\` or with `x:\` (where `x` is a drive letter), the path is relative to the current directory. Beware that \ must be written \\ in a string passed to the compiler.

```
Func FileDelete(name$[]|name${, opt%});
```

`name$` This is either a string variable or an array of strings that holds the names of the files to delete. Only one name per string and no wildcard characters are allowed. If the names do not include a path they refer to files in the current directory.

`opt%` If this is present and non-zero, the user is asked before each file in the list is deleted. You cannot delete protected or hidden or system files.

Returns The number of files deleted or a negative error code.

See also: `FilePath$(), FilePathSet(), FileList()`

FileExportAs()

This function saves the current data view or the sampling configuration as a file on disk. A data view is saved either in its native format, or as text or as a picture. It is equivalent to the two File menu commands Export As and Save configuration. This cannot be used for external text or binary files as they are already on disk.

```
Func FileExportAs(name${, type% {,yes%, {text$}}});
```

`name$` The name to use for saving. If the string is empty or if the string holds wild card characters * or ?, then the File menu Save As dialog opens, otherwise it is used directly. In Windows, the wildcards select the initial list of files. If the string is used directly, a default file extension is not provided; you must provide the extension yourself.

`type%` The type to save the file as (if omitted, type -1 is used):

- -1 Export in the native format for the data view. This is equivalent to using `type%` 0 for file and memory views or 12 for XY views.
- 0 Export part of the data view as set by `ExportFrameList()`, `ExportTimeRange()` and `ExportChanList()` to a new Signal data file. The file extension should be .cfs.
- 1 Save the contents of the current data or XY view as a text file. Signal saves the data as set by `ExportFrameList()`, `ExportTimeRange()` and `ExportChanList()` in the text format set by `ExportChanFormat() and ExportTextFormat()`. The file extension should be .txt.
- 5 Save data or XY view as a picture file. The file extension should be .wmf.
- 6 Save the sampling configuration in a configuration file. The file extension should be .sgc.
- 12 For XY views only, save as an XY data file. The file extension should be .sxy.
- 13 Save as a bitmap file. The file extension should be .bmp.

`yes%` If this operation would overwrite an existing file you are asked if you wish to do this unless `yes%` is present and non-zero. While an existing file is open in Signal you will not be able to overwrite it.

`text$` An optional prompt displayed as part of the file dialog to prompt the user.

Returns The function returns 0 if the operation was a success, or a negative error code.

See also: `EditCopy(), ExportChanFormat(), ExportChanList(), ExportFrameList(), ExportTimeRange(), ExportTextFormat()`

FileGetIntVar()

This function reads a CFS file variable of integer type from the file attached to the current view, which must be a file view. The CFS supports the use of file and frame variables of integer, floating point and string types. Software other than Signal may have included these when creating a data file.

```
Func FileGetIntVar(name$ {&nVar%{, &units$, {nType%}}});
```

`name$` The name of the variable to look for. This string is not case sensitive but every character including spaces must match exactly.

`nVar%` If present this returns the variable number, -1 if not found, or a negative error code.

`units$` If present this returns the units for the variable.

`nType%` If present this returns a code for the CFS type of an integer variable:

0: INT1, 1: WRD1, 2: INT2, 3: WRD2, 4: INT4:

Returns The function returns the value of the variable if the operation was a success, otherwise zero.

See also: `FileGetRealVar(), FileGetStrVar$(), FileVarCount(), FileVarInfo(), FrameGetIntVar(), FrameGetRealVar(), FrameGetStrVar$(), FrameVarCount(), FrameVarInfo()`

FileGetRealVar()

This function reads a CFS file variable of real type from the file attached to the current view, which must be a file view.

```
Func FileGetRealVar(name$ {&nVar%{, &units$}});
```

`name$` The name of the variable to look for. This string is not case sensitive but every character including spaces must match exactly.

`nVar%` If present this returns the variable number, -1 if not found, or a negative error code.

`units$` If present this returns the units for the variable.

Returns The function returns the value of the variable if the operation was a success, otherwise zero.

See also: `FileGetIntVar(), FileGetStrVar$(), FileVarCount(), FileVarInfo(), FrameGetIntVar(), FrameGetRealVar(), FrameGetStrVar$(), FrameVarCount(),FrameVarInfo()`

FileGetStrVar$()

This function reads a CFS file variable of string type from the file attached to the current view, which must be a file view.

```
Func FileGetStrVar$(name$ {&nVar%{, &units$}});
```

`name$` The name of the variable to look for. This string is not case sensitive but every character including spaces must match exactly.

`nVar%` If present this returns the variable number, -1 if not found, or a negative error code.

`units$` If present this returns the units for the variable.

Returns The function returns a string contents of the variable if the operation was a success, otherwise an empty string.

See also: `FileGetIntVar(), FileGetRealVar(), FileVarCount(), FileVarInfo(), FrameGetIntVar(), FrameGetRealVar(), FrameGetStrVar$(), FrameVarCount(),FrameVarInfo()`

FileList()

This function gets lists of files and sub-directories (folders) in the current directory and can also return the path to the parent directory of the current directory. This function can be used to process all files of a particular type in a particular directory.

```
Func FileList(names$[]|&name$, type%{, mask$});
```

`name$` This is either a string variable or an array of strings that is returned holding the name(s) of files or directories. Only one name is returned per string.

`type%` This sets the type of the objects to return information on. Allowed values are:

- -3 The parent directory of the current directory. The full path is returned.
- -2 Sub-directories of the current directory. No path is returned.
- -1 All files in the current directory.
- 0 Signal data files (`*.cfs`).
- 1 Text files (`*.txt`).
- 2 Output sequence files (`*.pls`).
- 3 Signal script files (`*.sgc`).
- 6 Signal configuration files (`*.sgc`).
- 12 XY view data file (`*.sxy`).

`mask$` This optional string limits the names returned to those that match it; * and ? in the mask are wildcards. ? matches any character and * matches any 0 or more characters. Matching is case insensitive and from left to right.

Returns The number of names that met the specification or a negative error code. This can be used to set the size of the string array required to hold all the results.

See also: `FilePath$(), FilePathSet(), FileDelete(), FileName$()`

FileName$()

This returns the name of the data file associated with the current view (if any). You can recall the entire file name, or any part of it. If there is no file the result is an empty string.

```
Func FileName$({mode%});
```

`mode%` If present, determines what to return, if omitted taken as 0.

- 0 Or omitted, returns the full file name including the path
- 1 The disk drive/volume name
- 2 The path section, excluding the volume/drive and the name of the file
- 3 The file name up to and not including the last dot in the name, excluding any trailing number
- 4 Any trailing numbers from 3
- 5 The end of the file name from the last dot

Returns A string holding the requested name, or a blank string if there is no file.

See also: `FileList(), FilePath$(), FilePathSet(), FileDelete()`

FileNew()

This is equivalent to the File menu New command. It creates a new window, also called a view, and returns the handle. You can create visible or invisible windows. Creating an invisible window lets you set the window position and properties before you draw it. The new window is the current view and if visible, the front view. Use `FileSaveAs()` to name created files.

```
Func FileNew(type%{, mode%);
```

`type%` The type of file to create:

- 0 A Signal data file based on the sampling configuration, ready for sampling. This opens a new file view which is also referred to as the sampling document view. It may also open other windows which will include the sampling control panels.
- 1 A text file in a window.
- 2 An output sequence file in a window.
- 3 A Signal script file in a window.
- 12 An XY view with one (empty) data channel. Use `XYAddData()` to add more data and `XYSetChan()` to create new channels.

`mode%` This optional argument determines how the new window is opened. The value is the sum of these flags. If the argument is omitted, its value is 0. The flags are:

- 1 Make the new window(s) visible immediately. If this flag is not set the window is created, but is invisible.
- 2 For data files, if the sampling configuration holds information for creating additional windows, use it. If this flag is not set, data files extract enough information from the sampling configuration to set the sampling parameters for the data channels.

Returns It returns the view handle (or the handle of the lowest numbered duplicate for a data file with duplicate windows) or a negative error.

See also: `FileOpen(), FileSave(), FileSaveAs(), FileClose(), SetMemory(), SampleStart(), XYAddDAta(), XYSetChan()`

FileOpen()

This is the equivalent of the File Open... menu command. It opens an existing Signal data file or a text file in a window, or an external text or binary file. If the file is already opened, a handle for the existing view is returned. The window becomes the new current view. You can create windows as visible or invisible. It is often more convenient to create an invisible window so you can position it before making it visible.

```
Func FileOpen(name$, type% {,mode% {,text$}});
```

`name$` The name of the file to open. This can include a path. The file name is operating system dependent, see `FileDelete()`. If the name is blank or holds wild card characters (Windows only), the file dialog opens for the user to select a file.

`type%` The type of the file to open. The types currently defined (see `ViewKind()`) are:

0 Open a Signal data file. A new **file view** is created.
1 Open a text file. A new **text view** is created.
2 Open an output sequence file. A new **output sequence view** is created.
3 Open a Signal script file. A new **script view** is created.
6 Load configuration file. No new view is created.
8 An external text file without a window. An invisible **external text view** is created in which `Read()` or `Print()` can be used.
9 An external binary file without a window. An invisible **external binary view** is created in which `BRead()`, `BWrite()`, `BSeek()` and other binary routines can be used.
12 Open an XY data file. A new **XY view** is created.

`mode%` This optional argument determines how the window or file opens. If the argument is omitted, its value is 0.

For file types 0 to 3 and 12 the value is the sum of:

1 Make the new window(s) visible immediately. If this flag is not set the window is created, but is invisible.

2 Read resource information associated with the file. This may create more than one window, depending on the file type. For data files, it restores the file to the state as it was closed. If the flag is unset, resources are ignored.

4 Return an error if the file is already open in Signal. If this flag is not set and the file is already in use, it is brought to the front and its handle is returned.

When used with file types 8 and 9 the following values of `mode%` are used. The file pointer (which sets the next output or input operation position) is set to the start of the file in modes 0 and 1 and to the end in modes 2 and 3.

0 Open an existing file for reading only.
1 Open a new file (or replace an existing file) for writing (and reading).
2 Open an existing file for writing (and reading) .
3 Open a file for writing (and reading). If the file doesn't exist, create it.

`text$` An optional prompt displayed as part of the file dialog, for all except type 6. If this is supplied then the file dialog will appear even if a complete file name is also supplied.

Returns If a file opens without any problem, the return value is the view handle for the file (if multiple views open, it is the handle for the first file view created). For configuration files (`type%` of 6), the return value is 0 if no error occurs. If the file could not be opened, or the user pressed Cancel in the file open dialog, the returned value is a negative error code.

If multiple windows are created for a data file, you can get a list of the associated view handles using `ViewList(list%[],64)`.

See also: `FileDelete(), FileNew(), FileSave(), FileSaveAs(), FileClose(), BRead(), BReadSize(), BSeek(), BWrite(), BWriteSize(), ViewFind(), ViewList(), ViewKind()`

FilePath$()

This function gets the "current directory", the place on disk where file open and file save dialogs start from. It can also get the path for created data files or the directory where the Signal application is installed.

```
Func FilePath$({opt%});
```

`opt%` If 0 or omitted, this gets the current directory. If 1, it returns the path for temporary sampled data files from the preferences dialog. If 2, it returns the path to the directory where the Signal application is installed. If 3, it returns the path for automatic file saving from the sampling configuration.

Returns A string holding the path or an empty string if an error is detected.

See also: `FilePathSet(), FileList(), FileName$()`

FilePathSet()

This function sets the "current directory" and the directory where Signal data files created by `FileNew()` are stored until they are saved using `FileSaveAs()`. It can also be used to create the directories.

```
Func FilePathSet(path${,opt%{, make%|text$}});
```

`path$` A string holding the new path to the directory. The path must conform to the rules for pathnames on the host system and be less than 255 characters long. If the path is empty a dialog opens for the user to select an existing directory.

`opt%` If omitted or zero, this sets the current directory. If 1, it sets the path for temporary sampled data files in the preferences dialog. If 3, it sets the path for automatic file saving in the sampling configuration.

`make%` If this is present and non-zero, the command will create the directory if all elements of the path exist except the last. You cannot use this option if `path$` is blank.

`text$` Optional prompt for use with the dialog.

Returns Zero if the path was set, or a negative error code.

See also: `FilePath$(), FileList()`

FilePrint()

This function is equivalent to the File menu Print command. It prints some or all of the current view to the printer that is currently set for Signal. If no printer has been set, the current system printer is used. In a file or memory view, it prints a range of data with the x axis scaling set by the display. In a text or log view, it prints a range of text lines. There is currently no script mechanism to choose a printer; you must do it interactively.

```
Func FilePrint({from{, to{, flags%}}});
```

`from` The start point of the print. This is in seconds in a file or memory view and in lines in a text view. If omitted, this is taken as the start of the view.

`to` The end point in the same units as from. If omitted or set beyond the end of the view then the end of the view is used.

`flags%` 0=portrait, 1=landscape, 2=current setting. If omitted, the current value is used.

Returns The function returns 0 if all went well; otherwise it returns a negative error.

The format of the printed output is based on the screen format of the current view. Beware that for file and memory views the output could be many (very many) pages long.

See also: `FilePrintScreen()`, `FilePrintVisible()`

FilePrintScreen()

This function is equivalent to the File menu Print screen command. It prints all visible data, text, script, sequence, XY and log views on the screen to the current printer on one page. The page positions are proportional to the view positions on the screen.

```
Func FilePrintScreen({pTtl${, vTtl%{, box%{, scTxt%(
                                              ,flags%}}}}});
```

`pTtl$` This sets the page title string to print at the top of a page. If omitted or an empty string, there is no page title.

`vTtl%` Set 1 or higher to print a title above each view, omitted or 0 for no title.

`box%` Set 1 or higher for a box around each view. If omitted, or 0, no box is drawn.

scTxt% Set 1 or higher to scale text differently in the x and y directions to match the original. If omitted or 0 scale both directions by the same scale factor.

`flags%` 0=portrait, 1=landscape, 2=current setting. If omitted, the current value is used. Prior to version 2.13, the default was landscape mode.

Returns The function returns 0 if it all went well, or a negative error code.

See also: `FilePrint()`, `FilePrintVisible()`

FilePrintVisible()

This function prints the current view as it appears on the computer screen to the current printer. In a text view, this prints the lines in the current selection. If there is no selection, it prints the line containing the cursor. This function is equivalent to the File menu Print visible command.

```
Func FilePrintVisible({flags%});
```

`flags%` 0=portrait, 1=landscape, 2=current setting. If omitted, the current value is used.

Returns The function returns 0 if all went well, otherwise it returns a negative error.

See also: `FilePrint()`, `FilePrintScreen()`

FileQuit()

This is equivalent to the File menu Exit command. If there is any unsaved data you are asked if you wish to save it before the application closes. If the user cancels the operation (because there were files that needed saving), the script terminates, but the Signal application is left running. Use `FileClose(-1, -1)` before `FileQuit()` to guarantee to exit.

```
Proc FileQuit();
```

See also: `FileClose()`

FileSave()

This function saves the current view as a file on disk. It is equivalent to the File menu Save command. You cannot use this command for a file view if it has just been sampled, use `FileSaveAs()` instead. If the view has not been saved previously, the File menu Save As dialog opens and the user must provide a file name. This cannot be used for external text or binary files either as they are already on disk.

```
Func FileSave();
```

Returns The function returns 0 if the operation was a success, or a negative error code.

See also: `FileOpen(), EditCopy(), FileExportAs(), FileSaveAs(), FileClose()`

FileSaveAs()

This function is equivalent to the File menu SaveAs command. It can be used to save the current view, with any changes, under a new name. Use this to save and name a Signal data file immediately after it has been sampled. Use `FileExportAs()` to export selected parts of a CFS data file to a new file, or to export from a view under a different format.

```
Func FileSaveAs(name$ {,yes%, {text$}});
```

`name$` The name to use for saving. If the string is empty or if the string holds wild card characters * or ?, then the File menu Save As dialog opens. In Windows, the wildcards select the initial list of files. A default extension is not provided in circumstances when the dialog is not used.

`yes%` If this operation would overwrite an existing file you are asked if you wish to do this unless `yes%` is present and non-zero. While an existing file is open in Signal you will not be able to overwrite it.

`text$` An optional prompt displayed as part of the file dialog to prompt the user.

Returns The function returns 0 if the operation was a success, or a negative error code.

See also: `FileSave(), EditCopy(), FileExportAs()`

FileVarCount()

This function counts CFS file variables in the data file.

```
Func FileVarCount();
```

Returns The number of file variables in the data file associated with this view.

See also: `FileGetIntVar(), FileGetRealVar(), FileGetStrVar$(), FileVarInfo(), FrameGetIntVar(), FrameGetRealVar(), FrameGetStrVar$(), FrameVarCount(), FrameVarInfo()`

FileVarInfo()

This function reads a description of a CFS file variable.

```
Func FileVarInfo(nVar%, &name$);
```

`nVar%` This is the variable number.

`name$` The name of the variable, which can be used in the commands for reading the file variables.

Returns The function returns the type of the variable or -1 if the variable was not found or is of unknown type. The type code is as follows:

0 An integer variable which can be read using `FileGetIntVar()`
1 A floating point variable which can be read using `FileGetRealVar()`
2 A string variable which can be read using `FileGetStrVar$()`

See also: `FileGetIntVar()`, `FileGetRealVar()`, `FileGetStrVar$()`, `FileVarCount()`, `FrameGetIntVar()`, `FrameGetRealVar()`, `FrameGetStrVar$()`, `FrameVarCount()`, `FrameVarInfo()`

FiltApply()

Applies a set of filter coefficients or a filter in the filter bank to a set of waveform channels in the current file or memory view.

Each output point is generated from the same number of input points as there are filter coefficients. Half these points are before the output point, and half are after. Where more data is needed than exists in the source file (for example at the start and end of a file and where there are gaps), extra points are made by duplicating the nearest valid point.

```
Func FiltApply(n%|coef[], cSpc, frm%|frm%[]|frm$);
```

`n%` Index of the filter in the filter bank to apply in the range 0-11, or

`coef[]` An array holding a set of FIR filter coefficients to apply to the waveform.

`cSpc` A channel list specifier of the channels to filter. See the *Script language syntax* chapter for a definition of channel specifiers.

`frm%` Frame number or a negative code as follows:

-1 All frames in the file
-2 The current frame
-3 Only tagged frames
-6 Only untagged frames

`frm$` A frame specification string. This option specifies a list of frames using a string such as “1..32,40,50”.

`frm%[]` An array of frame numbers. This option provides a list of frame numbers. The first element holds the number of frames in the list.

Returns The number of the last channel to be filtered or a negative error code. A negative error code is also returned if the user clicks Cancel from the progress bar that may appear during a long filtering operation.

See also: `FiltAtten()`, `FiltCalc()`, `FiltComment$()`, `FiltCreate()`, `FiltInfo()`, `FiltName$()`, `FiltRange()`

FiltAtten()

This set the desired attenuation for a filter in the filter bank. When `FiltApply()` or `FiltCalc()` is used, the number of coefficients needed to achieve this attenuation (up to a maximum of 255) will be generated. A value of zero sets the attenuation back to the default (-65 dB).

```
Func FiltAtten(index%{, dB});
```

`index%` Index of the filter in the filter bank to use in the range 0-11.

`dB` If present and negative, this is the desired attenuation for stop bands in the filter.

Returns The desired attenuation for a filter at the time of the call.

See also: `FiltApply(), FiltCalc(), FiltComment$(), FiltCreate(), FiltInfo(), FiltName$(), FiltRange()`

FiltCalc()

The calculation of filter coefficients can take an appreciable time. This routine forces the calculation of a filter for a particular sampling frequency if it has not already been done. If you do not force the calculation, you can still use `FiltApply()` to apply a filter. However, the coefficient calculation will then be done at the time of filter application, which may not be desirable if the filtering operation is time critical.

```
Func FiltCalc(index%, sInt{, coeff[]{, &dBGot {, nCoef%}}});
```

`index%` Index of the filter in the filter bank to use in the range 0-11.

`sInt%` The sample interval of the waveform you are about to filter. This is the value returned by `BinSize()` for a waveform channel.

`coeff` An array to be filled with the coefficients used for filtering. If the array is too small, as many elements as will fit are set. The maximum size needed is 255.

`dBGot` If present, returns the attenuation attained by the filter coefficients.

`nCoef%` If present, sets the number of coefficients used in the calculation (use an even number for a full differentiator and an odd number for all other filter types).

Returns The number of coefficients generated by the filter.

An example

Suppose the first filter in the bank (index 0) is a low pass filter with the pass band edge at 50 Hz. If we know that we will need to filter a channel 4 (sampled at 200 Hz) with this filter, we may want to calculate the coefficients needed in advance:

```
FiltCalc(0, BinSize(4));
```

This will calculate a filter corresponding to the specification of filter 0 for a sampling frequency of 200 Hz with an attenuation in the stop band of at least the current desired attenuation value for this filter.

Constraints on filters

The calculation of coefficients is a complex process and can produce silly results due to floating point rounding errors in some situations. To ensure that you will always get a useful result there is a limit to how small and how big a transition gap can be, relative to the sampling frequency. There is a similar limit on the width of a pass or stop band:

- The transition gap and the width of a pass or stop band cannot be smaller than 0.005 of the sampling frequency.
- The transition gap cannot be larger than 0.12 of the sampling frequency.

This function always calculates a set of coefficients, but may alter the filter specification in order to do it (these changes are temporary, see later.) This can happen in two cases :

1. If the sampling frequency is such that, to produce the filter, the transition gap and/or pass and stop band widths are outside their limits, then the widths are set to the limits before calculating the filter. In our 50 Hz low pass filter example, if we calculate it with respect to a 12 kHz sampling frequency, the minimum pass band width is 12000*0.005 = 60 Hz. So, the filter would be changed to a 60 Hz low pass filter.
2. If half the sampling frequency (the Nyquist frequency) is less than an edge of a pass or stop band, certain attributes of the filter are lost. In our 50 Hz low pass filter example, if we tried to calculate with a sampling frequency of 80 Hz, we would see that the Nyquist frequency is 40 Hz. No frequency above 40 Hz can be represented in a waveform sampled at 80 Hz, so a 50 Hz low pass filter is equivalent to an "All pass" filter. The filter specification will be altered to reflect this before calculating.

Any changes made to a filter specification to accommodate a particular calculation are made with reference to the original specification, not the specification that was last used for a calculation.

See also: `FiltApply()`, `FiltAtten()`, `FiltComment$()`, `FiltCreate()`, `FiltInfo()`, `FiltName$()`, `FiltRange()`

FiltComment$()

This function gets and sets the comment associated with a filter in the filter bank.

```
Func FiltComment$(index% {, new$});
```

`index%` Index of the filter in the filter bank to use, in the range `0-11`.

`new$` If present, sets the new comment.

Returns The previous comment for the filter at the index.

See also: `FiltApply()`, `FiltAtten()`, `FiltCalc()`, `FiltCreate()`, `FiltInfo()`, `FiltName$()`, `FiltRange()`

FiltCreate()

This function creates a filter in the filter bank to the supplied specification and gives it a standard name and comment.

```
Func FiltCreate(index%, type%{, trW{, edge1{, edge2{, ...}}}});
```

`index%` Index of the new filter in the filter bank in the range 0-11. This action replaces any existing filter at this index.

`type%` The type of the filter desired (see table).

`trW` The transition width of the filter. This is the frequency interval between the edge of a stop band and the edge of the adjacent pass band.

`edgeN` This is a list of edges of pass bands in Hz. (see table).

Returns 0 if there was no problem or a negative error code if the filter was not created.

This table shows the relationship between different filter types and the meaning of the corresponding arguments. The numbers in brackets indicate the nth pass band when there is more than 1. An empty space in the table means that the argument is not required.

type%	Name	trW	edge1	edge2	edge3	edge4
0	All stop					
1	All pass					
2	Low pass	Yes	High			
3	High pass	Yes	Low			
4	Band pass	Yes	Low	High		
5	Band stop	Yes	High(1)	Low(2)		
6	Low pass differentiator	Yes	High			
7	Differentiator					
8	1.5 Band Low pass	Yes	High(1)	Low(2)	High(2)	
9	1.5 Band High pass	Yes	Low(1)	High(1)	Low(2)	
10	2 Band pass	Yes	Low(1)	High(1)	Low(2)	High(2)
11	2 Band stop	Yes	High(1)	Low(2)	High(2)	Low(3)

The values entered correspond to the text fields shown in the Filter edit dialog box.

See also: `FiltApply(), FiltAtten(), FiltCalc(), FiltComment$(), FiltInfo(),FiltName$(), FiltRange()`

FiltInfo()

Retrieves information about a filter in the bank.

```
Func FiltInfo(index%{, what%});
```

`index%` Index of the filter in the filter bank to use in the range 0-11.

`what%` Which bit of information about the filter to return:

- -2 Maximum `what%` number allowed
- -1 Desired attenuation
- 0 type (if you supply no value, 0 is assumed)
- 1 Transition width
- 2-5 `edge1-edge4` given in `FiltCreate()`

Returns The information requested as a real.

See also: `FiltApply(), FiltAtten(), FiltCalc(), FiltComment$(), FiltCreate(), FiltName$(), FiltRange()`

FiltName$()

This function gets and/or sets the name of a filter in the filter bank.

```
Func FiltName$(index% {, new$});
```

`index%` Index of the filter in the filter bank to use in the range 0-11.

`new$` If present, sets the new name.

Returns The previous name of the filter at that index.

See also: `FiltApply(), FiltAtten(), FiltCalc(), FiltComment$(), FiltInfo(), FiltCreate(), FiltRange()`

FiltRange()

Retrieves the minimum and maximum sampling rates that this filter can be applied to without the specification being altered. See the `FiltCalc()` command, *Constraints on filters* for more information.

```
Proc FiltRange(index%, &minFr, &maxFr);
```

`index%` Index of the filter in the filter bank to use in the range 0-11.

`minFr` Returns the minimum sampling frequency you can calculate the filter with respect to, so that no transition width is greater than the maximum allowed and no attributes of the filter are lost.

`maxFr` Returns the maximum sampling frequency you can calculate the filter with respect to, without the transition (or band) widths being smaller than allowed.

It is possible to create a filter which cannot be applied to any sampling frequency without being changed. This will be apparent because `minFr` will be larger than `maxFr`.

See also: `FiltApply()`, `FiltAtten()`, `FiltCalc()`, `FiltComment$()`, `FiltInfo()`, `FiltCreate()`, `FiltName$()`

FIRMake()

This function creates FIR filter coefficients and places them in an array ready for use by `ArrFilt()`. This command is very similar in operation to the DOS program `FIRMake` and has similar input requirements. Unless you need precise control over all aspects of filter generation, you may find it easier to use `FiltCalc()` or `FIRQuick()`. You will need to read the detailed information about FIR filters in the description of the Digital Filter dialog to get the best results from this command.

```
Proc FIRMake(type%, param[][], coef[]{, nGrid{, extFr[]}});
```

`type%` The type of filter file to produce: 1=Multiband filter, 2=Differentiator, 3=Hilbert transformer, 4=Multiband pink noise (Multiband with 3 dB per octave roll-off).

`param` This is a 2-dimensional array. The size of the first dimension must be 4 or 5. The size of the second dimension (`n`) should be the number of bands in your filter. You pass in 4 values for each band (indices 0 to 3) to describe your filter:

Indices 0 and 1 are the start and end frequency of each band. All frequencies are given as fraction of a sampling frequency and so are in the range 0 to 0.5.

Index 2 is the function of the band. For all filter types except a differentiator, this is the gain of the filter in the band in the range 0 to 1 (the most common values are 0 for a stop band and 1 for a pass band.) For a differentiator, this is the slope of the filter in the band, normally not more than 2. The gain at any frequency *f* in the band is given by *f*function*.

Index 3 is the relative weight to give the band. The weight sets the relative importance of the band in multiband filters. The program divides each band into frequency points and optimises the filter such that the maximum ripple times the weight in each band is the same for all bands. The weight is independent of frequency, except in the case of the differentiator, where the weight used is weight/frequency.

If there is an index 4 (the size of the first dimension was 5), this index is filled in by the function with the ripple in the band in dB.

`coef` An array into which the FIR filter coefficients are placed. The size of this array determines the number of filter coefficients which are calculated. It is important, therefore, to make sure this array is exactly the size that you need. The maximum number of coefficients is 256.

`nGrid` The grid density for the calculation. If omitted or set to 0, the default density of 16 is used. This sets the density of test points in internal tables used to search for points of maximum deviation from the filter specification. The larger the value, the longer it takes to compute the filter. There is seldom any point changing this value unless you suspect that the program is missing the peak points.

`extFr` An array to hold the list of extremal frequencies (the list of frequencies within the bands which have the largest deviation from the desired filter). If there are `n%` coefficients, there are `(n%+1)/2` extremal frequencies.

The parameters passed in must be correct or a fatal error results. Errors include: overlapping band edges, band edges outside the range 0 to 0.5, too many coefficients, differentiator slope less than 0. If not a differentiator the band function must lie between 0 and 1, the band weight must be greater than 0.

For example, to create a low pass filter with a pass band from 0 to 0.3 and a stop band from 0.35 to 0.5, and no return of the ripple, you would set up `param` as follows:

```
var param[4][2]      'No return of ripple, 2 bands
para[0][0] := 0;     'Starting frequency of pass band
para[1][0] := 0.3;   'Ending frequency of pass band
para[2][0] := 1;     'Desired gain (unity)
para[3][0] := 1;     'Give this band a weighting of 1

para[0][1] := 0.35;  'Starting frequency of stop band
para[1][1] := 0.5;   'Ending frequency of stop band
para[2][1] := 0;     'Desired gain of 0 (stop band)
para[3][1] := 10;    'Give this band a weighting of 10
```

See also: `ArrFilt()`, `FiltApply()`, `FiltCalc()`, `FIRQuick()`, `FIRResponse()`

FIRQuick()

This function creates a set of filter coefficients in the same way the `FIRMake()` does, but many of the parameters are optional, allowing the most common filters to be created with a minimal specification.

```
Func FIRQuick(coef[], type%, freq {, width {, atten}});
```

`coef` An array into which the FIR filter coefficients are placed. The size of this array should be 256. This is the maximum number of coefficients that can be created and this function reserves the right to return as many as it feels necessary, up to that value to create a decent filter.

`type%` This sets the type of filter to create. 0=Low pass, 1=High pass, 2=Band pass, 3=Band stop and 4=Differentiator.

`freq` This is a fraction of the sampling rate in the range 0 to 0.5 and means different things depending on the type of filter.:

For Low pass, High pass and Differentiator types, this represents the cut-off frequency. This is the frequency of the higher edge of the first frequency band.

For Band pass and Band stop filters, this is the midpoint of the middle frequency band: the pass band in a Band pass filter, the stop band in a Band stop filter.

`width` For Low pass, High pass and Differentiator filters, this is the width of the transition gap between the stop band and the pass band. The default value is 0.02 and there is an upper limit of 0.1 on this argument.

For a band Pass or Band stop filter, `width` is the width of the middle band. E.g. if you ask for a Pass band filter with the `freq` parameter to be 0.25 and the width to be 0.05, the middle pass band will be from 0.2 to 0.3. For these types of filter, you still need a positive transition width. This transition width is 0.02 and cannot be changed by the user.

`atten` The desired attenuation in the stop band in dB. The default is 50 dB. This is analogous to the desired attenuation in the `FiltAtten()` command.

Returns The number of coefficients calculated. If the array is not large enough the coefficient list is truncated (and the result is useless).

See also: `ArrFilt(), FiltApply(), FiltCalc(), FIRMake(), FIRResponse()`

FIRResponse()

This function retrieves the frequency response of a given filter as amplitude or in dB.

```
Func FIRResponse(resp[], coef[]{, as%{, type%}});
```

`resp` The array to hold the frequency response. This array will be filled regardless of its size. The first element is the amplitude response at 0 Hz and the last is the amplitude response at the Nyquist frequency. The remaining elements are set to the response at a frequency proportional to the element's position in the array.

`coef` The coefficient array calculated by `FIRMake()`, `FIRQuick()` or `FiltCalc()`.

`as%` If this is 0 or omitted, the response is in dB (0 dB is unchanged amplitude), otherwise as linear amplitude (1.0 is unchanged).

`type%` If present, informs the command of the filter type. The types are the same as those supplied for `FIRQuick()`: 0=Low pass, 1=High pass, 2=Band pass, 3=Band stop and 4=Differentiator. If a type is given , the time to calculate the response is halved. If you are not sure what type of filter you have, or you have type not covered by the `FIRQuick()` types, then do not supply a type to this command.

See also: `ArrFilt(), FiltCalc(), FIRMake(), FIRQuick()`

Fitting

It frequently happens that you have a set of data values (x_1,y_1), (x_2,y_2) … (x_n,y_n) that you wish to test against a theoretical model $y = f(x, \mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2\ldots)$ where the $\mathbf{a}_i$ are coefficients that are to be set to constant values which give the *best fit* of the model to the data values.

For example, if we were looking at the extension of a spring (y) as it is loaded by weights (x), we might wish to fit the straight line $y = \mathbf{a}_0 + \mathbf{a}_1 x$ to some measured data points so that we could measure a weight by the extension it caused. A careful experimenter might also wish to know what the probable error was in $\mathbf{a}_0$ and $\mathbf{a}_1$ so that the probable error in any weight deduced from an extension would be known. An even more cautious experimenter might want to know if the straight line formula was likely to model the measured data.

To avoid repeating definitions throughout the remainder of this chapter the following will be taken as defined. We apologise to the statisticians who may read the following and shudder:

mean

Given a set of n values y_j, the mean is $\Sigma_j\, y_j / n$ (the symbol Σ_j means, form the sum over all indices $_j$ of the expression that follows).

variance

If the mean of a set of n data values y_j is y_m, then the variance σ^2 (sigma squared) of this set of values is:

$\sigma^2 = \Sigma_j\, (y_j - y_m)^2 / n$ if y_m is known independently of the data values y_j

$\sigma^2 = \Sigma_j\, (y_j - y_m)^2 / (n - 1)$ if y_m is calculated from the data values y_j

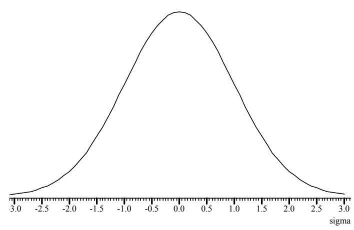
For a data set of any reasonable size, the use of n-1 or n in the denominator should make little difference.

standard deviation

The standard deviation σ (sigma) of a data set is the square root of the variance. Both the variance and the standard deviation are used as measures of the width of the distribution.

Normal distribution

If you measure a data value in any real system, there is always some error in the measurement. Once you have made a (very) large number of measurements, you can form a curve showing the probability of getting any particular value. One would hope that this error distribution would show a large peak at the "correct" value of the measurement and the width of this distribution would show the spread of likely errors.



There is a particular error distribution which often occurs, called the Normal distribution. If a set of measurements is normally distributed, with a mean y_m and standard deviation σ, then the probability of measuring any particular value y is proportional to:

$P(y) \propto \exp(-½(y-y_m)^2/\sigma^2)$

It is for this distribution of errors that we have the well-known result that 68% of the values lie within one standard deviation of the mean, that 95% lie within two standard deviations and that 99.7% lie within three standard deviations. Of course, **if the error distribution is not normal, these results do not apply**.

chi-squared The fitting routines given here define *best fit* as the values of $\mathbf{a}_i$ (the coefficients) that minimise the chi-squared value (χ^2), defined as the sum over the measured data points of the square of the difference between the measured and predicted values divided by the variance of the data point:

$$\chi^2 = \Sigma_j \left((y_j - f(x_j, \mathbf{a}_i))^2 / \sigma_j^2 \right)$$

where (x_j, y_j) is a data point and σ_j^2 is the variance of the measured data at that point.

If the sigma of each data point is unknown, then the fitting routines can be used to minimise $\Sigma_j (y_j - f(x_j, \mathbf{a}_i))^2$, which produces the same result as a chi-squared fit would produce if the variance of the errors at all the data points was the same. This is commonly called least-squares fitting, meaning that the fit minimises the sum of squares of the errors between the fitted function and the data.

Chi-squared fitting is also a maximum-likelihood fit if the errors in the data points are normally distributed. This means that as well as minimising the chi-squared value, the fit also selects the most probable set of coefficients that model your data. If your data measurement errors are not normally distributed you can still use this method, but the fit is not maximum likelihood.

If your errors are normally distributed and if you know the variance(s) of the data points, you can form good estimates of the variance of the fitted coefficients, and you can also test whether the function you have fitted is likely to model the data.

If your errors are normally distributed but you do not know the variance of the errors at the data points, you can make an estimate of the variance of the errors. (This is based on the assumption that the variance is the same for all of them, and that the model does fit the data.) First you fit your model, then you calculate the variance from the errors between the best fit and the data. Having done this, you *cannot* then use this variance to test if the fit is likely to model the data!

Residuals Once your fit is completed, it is a good idea to look at the graph of the errors between your original data and the fitted data (the residuals or residual errors). If your errors are normally distributed and are independent, you would expect this graph to be more or less horizontal with no obvious trends. If this is not the case, you should consider whether the correct model function has been selected, or whether the fitting function has found the true minimum.

Linear fit
Non-linear fit A linear fit is one in which the theoretical model $y = f(x, \mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2 \ldots)$ can be expressed as $y = \mathbf{a}_0 f_0(x) + \mathbf{a}_1 f_1(x) + \mathbf{a}_2 f_2(x) \ldots$ for example $y = \mathbf{a}_0 + \mathbf{a}_1 x + \mathbf{a}_2 x^2$. Linear fits are relatively quick as they are done in one step. Usually, the only thing that can cause a problem is if the functions $f_i(x)$ are not linearly independent. The methods we use can usually detect this problem, and can still give a useful result.

A non-linear fit means all other cases, for example, $y = \mathbf{a}_0 \exp(-\mathbf{a}_1 x) + \mathbf{a}_2$. These types of problem are solved by making an initial guess at the coefficients (and ideally providing a range of values that the result is known to lie in) and then improving this guess. This process repeats until some criterion is met. Each repeat is called an *iteration*, so we call this an iterative process.

Covariance array

Several of the fitting routines return a covariance array. If you have *n* coefficients, this array is of size *n* by *n* and is diagonally symmetric. If the errors in the original data points are normally distributed, the diagonal elements of this array are the variances in the values of the fitted coefficients. The remaining elements are the co-variances of pairs of the fitting parameters and can be used to estimate errors in derived values that depend on the product of two of the coefficients. If the errors are not normally distributed, the further away from normal the errors are, the less useful is the covariance array as a direct indication of the variances in the fitted coefficients.

For example, in the case of the linear fit $y = \mathbf{a}_0 + \mathbf{a}_1 x + \mathbf{a}_2 x^2$ you might collect your three coefficients in the array `coef[]`, and the covariance in the array `covar[][]`. In this case, the $\mathbf{a}_0$ value is returned in `coef[0]` and its variance in `covar[0][0]`, the $\mathbf{a}_1$ value is returned in `coef[1]` and its variance in `covar[1][1]`, and the $\mathbf{a}_2$ value is returned in `coef[2]` and its variance in `covar[2][2]`.

Because the array is diagonally symmetric, `covar[i][j]` is equal to `covar[j][i]` and the off-diagonal elements are the expected variance in the product of pairs of the coefficients, so `covar[1][2]` is the variance of $\mathbf{a}_1\mathbf{a}_2$.

If you have not supplied the standard deviations of the errors in the data points, the covariance array is calculated on the assumption that all data points have a standard deviation of 1.0, and the covariance array is incorrectly scaled. In this case, if inspection of the residuals leads you to the conclusion that the function does indeed fit the data and that the errors are more or less the same for all values and not too far from normally distributed, then you can scale the covariance array to the correct values. Multiply all the elements of the array by the sum of squares of the errors between the data and the fitted values divided by the number of data points. If there are `nD%` data points and the sum of squares of the errors is `errSq`, then use `ArrMul(cover[][], errSq/nD%)` to rescale the covariance.

What does the covariance mean?

Having fitted our data, we would like some idea of how the errors in the original data feed through to uncertainties in the values of the coefficients. The best way to do this is to obtain many sets of (x,y) data and fit our coefficients to each set. Then we can inspect the values of the coefficients and obtain a mean and standard deviation for each coefficient. However, this is very time consuming.

If the errors in the data are normally distributed (or not too far from this ideal case) and known, then the covariance array gives you some useful information. The square root of the covariance for a particular coefficient is the expected standard deviation in that value (given that the remaining coefficients remain fixed at optimum values). In script language terms, the standard deviation of `coef[i]` is `sqrt(covar[i][i])`.

In this case you would expect the coefficient to be within one standard deviation of the "correct" result 68% of the time, within 2 standard deviations 95% of the time and within 3 standard deviations 99.7% of the time.

Testing the fit

If the errors in the original data are normally distributed and known (not calculated from the fit), and you know the χ^2 value for the fitted data, you can ask the question, "Given the known errors in the original data, how likely is it that you would get a value of χ^2 at least this large?" The answer is (at least in terms of the script language) that the probability is: `GammaQ((nData% - nCoef%)/2.0, chiSq/2.0);` where `nData%` is the number of data points to be fitted, `nCoef%` is the number of coefficients that were fitted and `chiSq` is the χ^2 value for the fit. `GammaQ()` is the incomplete Gamma function.

If you want to follow this result up in a statistical textbook, you should look up *chi-squared distribution for n degrees of freedom*. In our case, we have `nData%-nCoef%` degrees of freedom.

If the fit is reasonable, you should expect a probability value between 0.1 and 1 (but be a bit suspicious if you always get values close to 1.0, as you may have overestimated the errors in the data.) If the wrong function has been fitted or if the fit is poor you usually get a very small probability. Intermediate values (0.0001 to 0.1) may indicate that the errors in the original data were actually larger than you thought, or they may indicate that the data just doesn't fit the model.

FitCoef()

This command gives you access to the fit coefficients for the next `FitData()` fit. You can return the values from any type of fit and set the initial values and limits and hold values fixed for iterative fits. There are two command variants:

Set and get coefficients

This command variant lets you read back the current coefficient values and set the coefficient values and limits for iterative fitting:

```
Func FitCoef({num%{, new{, lower{, upper}}}});
```

`num%` If this is omitted, the return value is the number of coefficients in the current fit. If present, it is a coefficient number. The first coefficient is number 0. If `num%` is present, the return value is the coefficient value for the existing fit, or if there is no fit, the coefficient value that would be used as the starting point for the next iterative fit is returned.

`new` If present, this sets the value of coefficient `num%` for the next iterative fit.

`lower` If present, this sets the lower limit for coefficient `num%` for the next iterative fit. There is currently no way to read back the coefficient limits. There is also no check made that the limits are set to sensible values.

`upper` If present, this sets the upper limit for coefficient `num%` for the next iterative fit.

Returns The number of coefficients or the value of coefficient `num%`.

Get and set the hold flags

This command variant allows you to hold some coefficients at their current values during the next fit.

```
Func FitCoef(hold%[]);
```

`hold%` An array of integers to correspond with the coefficients. If the array is too long, extra elements are ignored. If it is too short, extra coefficients are not affected. Set `hold%[i%]` to 1 to hold coefficient `i%` and to 0 to fit it. If `hold%[i%]` is less than 0, the hold state is not changed, but `hold%[i%]` is set to 1 if the corresponding coefficient is held and to 0 if it is not held.

Returns This always returns 0.

See also:`FitData()`, `FitValue()`, `FitExp()`, `ChanFitCoef()`

FitData()

This function, together with `FitCoef()` and `FitValue()`, lets you apply the same fitting functions that are available for channel data to data in arrays. You supply arrays of x and y data points and an optional array holding the standard deviation of the input data point y values. There are three command variants:

Initialise fit information

The first variant sets the type of fit. If you select an iterative fit, the initial values of the fitting coefficients are reset to standard values and any "hold" flags set by `FitCoef()` are cleared. You can set your own initial values with the `FitCoef()` command or make a guess at the initial values when performing the fit.

```
Func FitData(type%, order%);
```

`type%` The fit type. 0=Clear any fit, 1=Exponential, 2=Polynomial, 3=Gaussian, 4=Sine, 5=Sigmoid.

`order%` If positive, this is the order of the fit, if negative it is minus the number of fitted coefficients. See the information about each fit for the allowed values for each fit type. If `type%` is 0 this argument is ignored and should be 0.

Returns The number of fit coefficients for the fit or a negative error code.

Exponential fit

This fits multiple exponentials by an iterative method. The data is fitted to the equation:

$y = \mathbf{a}_0 \exp(-x/\mathbf{a}_1) + \mathbf{a}_2 \exp(-x/\mathbf{a}_3)\ldots$ For an even number of coefficients
$y = \mathbf{a}_0 \exp(-x/\mathbf{a}_1) + \mathbf{a}_2 \exp(-x/\mathbf{a}_3)\ldots + \mathbf{a}_n$ For an odd number of coefficients

You can set up to 10 coefficients or orders 1 to 5. If you use a fit order, the number of coefficients is the order times 2. See the `FitExp()` command for more information. Coefficient estimates are effective for orders 1 and 2.

Polynomial fit

This fits $y = \mathbf{a}_0 + \mathbf{a}_1 x + \mathbf{a}_2 x^2 + \mathbf{a}_3 x^3 \ldots$ to a set of (x,y) data points. The fitting is by a direct method; there is no iteration. The fit order is the highest power of x to fit in the range 1 to 10. The number of coefficients is the fit order plus 1.

Gaussian fit

This fits multiple Gaussians by an iterative method. The data is fitted to the equation:

$y = \mathbf{a}_0 \exp(-½(x-\mathbf{a}_1)^2/\mathbf{a}_2^2) + \mathbf{a}_3 \exp(-½(x-\mathbf{a}_4)^2/\mathbf{a}_5^2) + \ldots$

The fitted parameters (coefficients) are the $\mathbf{a}_i$. You can fit up to 3 Gaussians (order 1 to 3). The number of coefficients is given by the fit order times 3. Coefficient estimates become less useful as the order increases.

Sine fit

This fits multiple sinusoids by an iterative method. The data is fitted to the equation:

$y = \mathbf{a}_0 \sin(\mathbf{a}_1 x+\mathbf{a}_2) + \mathbf{a}_3 \sin(\mathbf{a}_4 x+\mathbf{a}_5) + \ldots \{+ \mathbf{a}_{3n}\}$

The fitted coefficients are the $\mathbf{a}_i$. Angles are evaluated in radians. You can fit up to 3 sinusoids (order 3) and an optional offset. Although the function is given in terms of sine functions, you can easily convert to cosines by subtracting π/2 from the phase angle ($\mathbf{a}_2$, $\mathbf{a}_5$, $\mathbf{a}_8$) after the fit. The coefficient count can be set to 3, 4, 6, 7, 9 or 10. If you use orders, the number of coefficients is order times 3 and you cannot set an offset. A useful coefficient estimate is made for a single sinusoid fit.

Sigmoid fit

This fits a single Boltzmann sigmoid by an iterative method. The data is fitted to the equation:

$y = \mathbf{a}_0 + (\mathbf{a}_1 - \mathbf{a}_0)/(1 + \exp((\mathbf{a}_2 - x)/ \mathbf{a}_3)$

In terms of the fitted result, $\mathbf{a}_0$ and $\mathbf{a}_1$ are the low and high fitting limits, $\mathbf{a}_2$ is the X50 point and $\mathbf{a}_3$ is a measure of the slope at the X50 point. You can set order 1 only, or 4 coefficients.

Perform the fit

This variant of the command does the fit set by the previous variant. Use the `FitCoef()` command to preset fit coefficients and to read back the result of the fit.

```
Func FitData(opt%, y[], x[]{, s[]|s{, &err{,
                                maxI% {,&iTer%{, covar[][]}}}}});
```

`opt%` 1=Estimate the coefficients before fitting, 0=use current values. Note that the estimates are usually only useful for a small number of coefficients.

`y[]` An array of y values to be fitted.

`x[]` A corresponding array of x values.

`s[]|s` A corresponding array of standard deviations for the data points defined by `y[]` and `x`, or a single value, being the standard deviation of each point. If this value is omitted or set to 1.0, the result is a least squares fit. If standard deviations are supplied, the result is a chi-squared fit.

`err` If present, this optional variable is updated with the chi-squared or least-squares error between the fit and the data.

`maxI%` If present, this changes the maximum number of iterations from 100.

`iTer%` If present, this integer variable is updated with the count of iterations done.

`covar` An optional two dimensional array of size at least `[nCoef][nCoef]` that is returned holding the covariance matrix when the fit is complete. It is changed if the return value is -1, 0 or 1. However, the values it contains are probably not useful unless the return value is 0.

Returns 0 if the fit is complete, 1 if max iterations done, or a negative error code: -1=the fit is not making progress (results may be OK), -2=the fit failed due to a singular matrix, -5=the fit caused a floating point error, -6=too little data for the number of coefficients, -7=unknown fitting function, -8=ran out of memory during the fit (too many data points), -9=the fit was set up with bad parameters.

Get fit information

This variant of the command returns information about the current fit.

```
Func FitData({opt%});
```

`opt%` This determines what to return. `opt%` values match `ChanFit()`, where possible. The returned information for each value of `opt%` is:

`opt%`	Returns	`opt%`	Returns
0	Fit type of next fit	1	Fit order of next fit
-1	1=a fit exists, 0=no fit exists	-8	`s` value used or 0 if an array
-2	Type of last fit or 0	-9	Not used
-3	Number of coefficients	-10	Not used
-4	Chi or least-squares error	-11	1=chi-square, 0=least-square
-5	Fit probability (estimated)	-12	Last fit result code
-6	Lowest x value fitted	-13	Number of fitted points
-7	Highest x value fitted	-14	Number of fit iterations used

Returns The information requested by the `opt%` argument or 0 if `opt%` is out of range.

See also:`ChanFit()`, `FitCoef()`, `FitValue()`

FitExp()

This command will fit multiple exponentials to arrays of x,y data points, with an optional weight for each point. Fitting is by an iterative method. The data is fitted to the equation:

$y = \mathbf{a}_0 \exp(-x/\mathbf{a}_1) + \mathbf{a}_2 \exp(-x/\mathbf{a}_3)\ldots$ For an even number of coefficients
$y = \mathbf{a}_0 \exp(-x/\mathbf{a}_1) + \mathbf{a}_2 \exp(-x/\mathbf{a}_3)\ldots + \mathbf{a}_n$ For an odd number of coefficients

The fitted parameters (coefficients) are the $\mathbf{a}_i$. You can fit up to 5 exponentials or 4 exponentials and an offset. However, experience shows that trying to fit more than two exponentials requires care. The fit from even two exponentials should be viewed with caution, especially if the exponential coefficients are similar. The odd numbered $\mathbf{a}_i$ are assumed to be positive. The commands to implement this are:

Set up the problem

The first command sets the number of exponentials to fit and the data set to be fitted. You must call this function before you call any of the others.

```
func FitExp(nCoef%, y[], x[]{, s[]|s});
```

`nCoef%` The number of coefficients to fit in the range 2 to 10. If this is even, the first form of the function above is used. If it is odd, the final coefficient is an offset.

`y[]` An array of y data values. The length of the array must be at least `nCoef%`.

`x[]` An array of x data values. The length of the array must be at least `nCoef%`.

`s` An optional argument which is either an array with one value for each y data point or a single value for all data points. If the value is the standard deviation, then the error value returned when you iterate to find the best fit is the chi-squared value and the fit is a chi-squared fit.

If this value is proportional to the error in the y values, then the fit is still a chi-squared fit, and the error returned is proportional to the chi-squared value. If you omit this array, the fit is a least squares error fit, and the error value returned is the sum of the squares of the errors in the y values.

Returns The function returns 0. There is no other return value as all errors stop the script.

The number of data points is set by the smallest of the sizes of the `y[]`, `x[]` or `s[]` (if present) arrays. The number of data points must be at least the number of coefficients. If it is not, you will get a fatal error, so check this before calling the function.

Set coefficient values and ranges

This variant of the function sets the initial value of each coefficient and optionally sets the range of allowed values. You can call this function at any time after the setup call and before the iterate call (below) has returned 0 indicating that the fit is completed.

In this type of fitting it is very important that you give reasonable starting values for the coefficients. In particular, when fitting multiple exponentials, you should try to limit the range of each exponent so that they cannot overlap. If you can do this, the fit will proceed quickly. If you do not give starting values, the command will make a simplistic guess at the fitting values. As we expect that you know more about the "right" answer than the command does, we suggest that you set the values you want.

```
func FitExp(coef%, val{, lo, hi});
```

`coef%` The coefficient to set. The first coefficient is number 0, the last one is `nCoef%`-1.

`val` The initial value to assign to the coefficient. If you have set low and high limits, and the value is outside these limits, it is set to the nearer limit.

`lo,hi` If present, these two values set the acceptable range of values for this coefficient. If omitted, or if both values are set to the same value, there is no limit. The value of the coefficient is tested against these limits after each iterative step, and if it exceeds a limit, it is set to that limit.

It is well worth limiting the exponent values so that they cannot be zero, which leads to degenerate cases. It is also worth limiting them so that they do not overlap as if two exponents get the same value the fit is degenerate and can wander around forever without getting anywhere. However, setting too rigid a range may damage the fitting process as sometimes the minimisation process has to follow a convoluted n-dimensional path to reach the goal, and the path may need to wander quite a bit. Let experience be your guide.

Iterate to a solution

Once you have set up the problem and given initial values to your coefficients, you can start the iteration process that will move the coefficients from their starting values to new values that minimise the error (optionally scaled by the `s` argument).

```
func FitExp(a[], &err{, maxI%{, &iTer%{, covar[][]}}});
```

`a[]` An array of size at least `nCoef%` that is returned holding the current set of coefficient values. The first amplitude is in `a[0]`, the first exponent in `a[1]`, the second amplitude in `a[2]`, the second exponent in `a[3]` and so on.

`err` A real variable returned as the sum over the data points of $(y_{\mathtt{x[i]}}-\mathtt{y[i]})^2/\mathtt{s[i]}^2$ if `s[]` is used or holding the sum of $(y_{\mathtt{x[i]}}-\mathtt{y[i]})^2$ if `s[]` is not used, where $y_{\mathtt{x[i]}}$ is the value predicted from the coefficients at the x value `x[i]`.

`maxI%` This is the maximum number of fitting iterations to do before returning from the function. If you omit this value, the function sets 1000. You can set any value from 1 to 10000. If you set more than 10000, the number is limited to 10000.

`iTer%` An optional integer variable that is returned holding the number of iterations done before the function returned.

`covar` An optional two dimensional array of size at least `[nCoef%][nCoef%]` that is returned holding the covariance matrix when the fit is complete. It is not changed unless the function return value is 0 or -1.

Returns This call returns 1 if the number of iterations has been completed, but the fitting process has not yet converged, 0 if the fitting process has converged, and a negative number if the fitting process is going nowhere.

Select coefficients to fit

Sometimes you will know the values of some of the coefficients, or you may wish to hold some coefficients fixed while you fit others. Normally the command fits all the coefficients, but you can use this command variant to select the coefficients to fit. You can use this command at any time after you have set the problem until the iteration variant returns 0 or -1.

```
func FitExp(fit%[]);
```

`fit%[]` An integer array of length at least `nCoef%`. Each element specifies if the corresponding coefficient is to be fitted (`fit%[i] := 1`) or held constant (`fit%[i] := 0`). If all elements are 0, then all arguments are fitted.

The effect of this command persists until either the iteration variant returns a value less than 1, or you set up a new problem, or you call this variant again.

An example

The following is a template for using this set of commands (assuming you don't want to hold some parameters constant).

```
const nData%:=50;          'set number of data elements
var x[nData%], y[nData%]; 'space for our arrays
var coef[4];              'space for coefficients
var err;                  'will hold error squared
...                       'in here goes code to get the data
FitExp(4, y[], x[]);      'fit two exponentials (no sigma array)
FitExp(0, 1.0, 0.2, 4);   'set first amplitude and limit range
FitExp(1, .01, .001, .03);'set first exponent and range
FitExp(2, 2.0, 0.1, 6);   'set second amplitude and limit range
FitExp(3, .08, .03, .15); 'set second exponent and range

repeat
   DrawMyData(coefs[], x[], y[]);  'Some function to show progress
until FitExp(coefs[], err, 1) < 1;

DrawMyData(coefs[], x[], y[]);     'Show the final state
```

See also: `ShowFunc()`

FitGauss()

This command will fit multiple gaussians to arrays of x,y data points, with an optional weight for each point. Fitting is by an iterative method. The input data is fitted to the equation:

$y = \mathbf{a}_0 \exp(-½(x-\mathbf{a}_1)^2/\mathbf{a}_2^2) + \mathbf{a}_3 \exp(-½(x-\mathbf{a}_4)^2/\mathbf{a}_5^2) + \ldots$

The fitted parameters (coefficients) are the $\mathbf{a}_i$. You can fit up to 3 gaussians. The commands to implement this are:

Set up the problem

The first command sets the number of gaussians to fit and the data set to be fitted. You must call this function before you call any of the others.

```
func FitGauss(nCoef%, y[], x[]{, s[]|s});
```

`nCoef%` The number of coefficients to fit. The only legal values are 3, 6 and 9 for one, two and three gaussians.

`y[]` An array of y data values. The length of the array must be at least `nCoef%`.

`x[]` An array of x data values. The length of the array must be at least `nCoef%`.

`s` An optional argument which is either an array with one value for each y data point or a single value for all data points. If the value is the standard deviation, then the error value returned when you iterate to find the best fit is the chi-squared value and the fit is a chi-squared fit.

If this value is proportional to the error in the y values, then the fit is still a chi-squared fit, and the error returned is proportional to the chi-squared value. If you omit this array, the fit is a least-squares error fit, and the error value returned is the sum of the squares of the errors in the y values.

Returns The function returns 0. There is no other return value as all errors stop the script.

The number of data points is set by the smallest of the sizes of the `y[]`, `x[]` or `s[]` (if present) arrays. The number of data points must be at least the number of coefficients. If it is not, you will get a fatal error, so check this before calling the function.

Set coefficient values and ranges

This variant of the function sets the initial value of each coefficient and optionally sets the range of allowed values. You can call this function at any time after the setup call and before the iterate call (below) has returned 0 indicating that the fit is completed.

In this type of fitting it is very important that you give reasonable starting values for the coefficients. In particular, when fitting multiple gaussians, it is usual that the centre of each distribution is easy to determine. If you can set the centres and limit them so that they cannot overlap, the fit usually will proceed without any problems, even for multiple gaussians. If you do not give starting values, the command will make a simplistic guess at the fitting values. As we expect that you know more about the “right” answer than the command does, we suggest that you set the values you want.

```
func FitGauss(coef%, val{, lo, hi});
```

`coef%` The coefficient to set. The first coefficient is number 0, the last one is `nCoef%`-1.

`val` The initial value to assign to the coefficient. If you have set low and high limits, and the value is outside these limits, it is set to the nearer limit.

`lo,hi` If present, these two values set the acceptable range of values for this coefficient. If omitted, or if both values are set to the same value, there is no limit. The value of the coefficient is tested against these limits after each iterative step, and if it exceeds a limit, it is set to that limit.

As long as you make a reasonable estimate of the centre points, there should be no problems fitting multiple gaussians.

Iterate to a solution

Once you have set up the problem and given initial values to your coefficients, you can start the iteration process that will move the coefficients from their starting values to new values that minimise the error (optionally scaled by the `s` argument).

```
func FitGauss(a[], &err{, maxI%{, &iTer%{, covar[][]}}});
```

`a[]` An array of size at least `nCoef%` that is returned holding the current set of coefficient values. The first amplitude is in `a[0]`, the first centre in `a[1]`, the first sigma in `a[2]`, the second amplitude in `a[3]` and so on.

`err` A real variable returned as the sum over the data points of $(y_{x[i]}-y[i])^2/s[i]^2$ if `s[]` is used or holding the sum of $(y_{x[i]}-y[i])^2$ if `s[]` is not used, where $y_{x[i]}$ is the value predicted from the coefficients at the x value `x[i]`.

`maxI%` This is the maximum number of fitting iterations to do before returning from the function. If you omit this value, the function sets 1000. You can set any value from 1 to 10000. If you set more than 10000, the number is limited to 10000.

`iTer%` An optional integer variable that is returned holding the number of iterations done before the function returned.

`covar` An optional two dimensional array of size at least `[nCoef%][nCoef%]` that is returned holding the covariance matrix when the fit is complete. It is not changed unless the function return value is 0 or -1.

Returns This call returns 1 if the number of iterations has been completed, but the fitting process has not yet converged, 0 if the fitting process has converged, and a negative number if the fitting process is going nowhere.

Remember that even when a minimum is found, there is no guarantee that this is *the* minimum. It is the best minimum that this algorithm can find given the starting point.

Select coefficients to fit Sometimes you will know the values of some of the coefficients, or you may wish to hold some coefficients fixed while you vary others. Normally the command will fit all the coefficients, but you can use this command variant to select the coefficient to fit. You can use this command at any time after you have set the problem until the iteration variant returns 0 or -1.

```
func FitGauss(fit%[]);
```

`fit%[]` An integer array of length at least `nCoef%`. Each element specifies if the corresponding coefficient is to be fitted (`fit%[i] := 1`) or held constant (`fit%[i] := 0`). If all elements are 0, then all arguments are fitted.

The effect of this command persists until either the iteration variant returns a value less than 1, or you set up a new problem, or you call this variant again.

An example The following is a template for using this set of commands (assuming you don't want to hold some parameters constant).

```
const nData%:=50;          'set number of data elements
var x[nData%], y[nData%]; 'space for our arrays
var s[nData%];             'space for sigma of each point
var coef[4];               'space for coefficients
var err;                   'will hold error squared
...                        'in here goes code to get the data
FitGauss(3, y[], x[], s[]); 'fit one gaussian
FitGauss(0, 1.0, 0.2, 4);   'set amplitude and limit range
FitGauss(1, 2, 1.5, 2.5);   'set centre of the gaussian and range
FitGauss(2, 0.5, 0.3, 1.9); 'set width and limit range
repeat
   DrawMyData(coefs[], x[], y[]);  'Some function to show progress
until FitGauss(coefs[], err, 1) < 1;
DrawMyData(coefs[], x[], y[]);     'Show the final state
```

See also: `ShowFunc()`

FitLine()

This function calculates the best fit line to a set of data points using the least squares error method. This can be applied to any Waveform channel. It fits the expression: $y = \mathbf{m}\ x + \mathbf{c}$ through the data points (x_i, y_i) so as to minimise the error given by: $\mathrm{Sum}_i(y_i - \mathbf{m}\ x_i - \mathbf{c})^2$. In this expression, **m** is the gradient of the line and **c** is the y axis intercept when x is 0.

```
Func FitLine(chan%, start, finish, &grad, &inter, &corr);
```

`chan%` A channel number (1 to n) holding waveform data.

`start` The start position for processing. `Start` and `finish` are given in x axis units.

`finish` The end position for processing. A data value at the `finish` position is included in the calculation.

`grad` This is returned holding the gradient of the best fit line (**m**).

`inter` This is returned holding the intercept of the line with the y axis (**c**).

`corr` This is returned holding correlation coefficient indicating the "goodness of fit" of the line. Values close to 1 or -1 indicate a good fit; values close to 0 indicate a very poor fit. This parameter is often referred to as *r* in textbooks.

Returns 0 if all was OK, or -1 if there were not at least 2 data points.

The results are in user units, so in a view with a waveform measured in volts on an x axis of seconds, the units of the gradient would be volts per second and the units of the intercept would be volts.

FitLinear()

This command fits $y = \mathbf{a}_0 f_0(x) + \mathbf{a}_1 f_1(x) + \mathbf{a}_2 f_2(x)$ … to a set of (x,y) data points. If you can provide error estimates for each y value, you can use the covariance output from this command to provide confidence limits on the calculated coefficients and you can use the returned chi-square value to test if the model is likely to fit the data. The command is:

```
func FitLinear(coef[], y[], x[][]{, s{, covar[][]{, r[]{, mR}}}});
```

`coef[]` A real array which sets the number of coefficients to fit and which returns the best fit set of coefficients. The array must be between 2 and 10 elements long. The coefficient $\mathbf{a}_0$ is returned in `coef[0]`, $\mathbf{a}_1$ in `coef[1]` and so on.

`y[]` A real array of y values.

`x[][]` This array specifies the values of the fitting functions at each data point. If there are *nc* coefficients and *nd* data values, this array must be of size at least [*nc*][*nd*]. If you think of this array as a rectangular grid with the data running from left to right and the coefficients running from top to bottom, the values you must fill in are:

$f_0(x_0)$	$f_0(x_1)$	$f_0(x_2)$	$f_0(x_3)$	…	$f_0(x_{n-1})$
$f_1(x_0)$	$f_1(x_1)$	$f_1(x_2)$	$f_1(x_3)$	…	$f_1(x_{n-1})$
$f_2(x_0)$	$f_2(x_1)$	$f_2(x_2)$	$f_2(x_3)$	…	$f_2(x_{n-1})$
$f_3(x_0)$	$f_3(x_1)$	$f_3(x_2)$	$f_3(x_3)$	…	$f_3(x_{n-1})$
…	…	…	…	…	…

`s` This is an optional argument. It is either a real array holding the standard deviations of each of the `y[]` data points, or it is a real value holding the standard deviation of all of the data points. If the argument is omitted or set to zero, a least squares error fit is performed, otherwise a chi-squared fit is done.

`covar` An optional two dimensional array of size at least [*nc*][*nc*] (*nc* is the number of coefficients fitted) that is returned holding the covariance matrix.

`r[]` An optional array of size at least [*nc*] (*nc* is the number of coefficients fitted) that is returned holding diagnostic information about the fit. The less relevant a fitting function $f_i(x)$ is to the fit, the smaller the value returned. The element of the array that corresponds to the most relevant function is returned as 1.0, smaller numbers indicate less relevance.

It can also sometimes happen that some of your base fitting-functions are not independent of each other, usually leading to huge coefficients that cancel each other out. In this case, several coefficients may be marked as of low relevance. The solution here is to remove one of the functions from the fit, or to set the next optional argument to exclude one of the functions, then fit again. If the remaining arguments become relevant, you have probably excluded a function that could be generated by a linear combination of the other functions. If the remaining arguments still are not relevant, you have eliminated a function that did not contribute to the fit.

`mR` You can use this optional variable to set the minimum relevance for a function. Functions that have less relevance than this are "edited" out of the fit and their coefficient is returned as 0. If you do not provide this value, the minimum is set to 10^{-15}, which will probably not exclude any values.

Returns The function returns the chi-square value for the fit if `s[]` or `s` is given (and non-zero), or the sum of squares of the errors between the data points and the best fit line if `s` is omitted or is zero.

The smallest of the sizes of the `y[]` array (and `s[]` array, if provided) and the second dimension of `x[][]` sets the number of data points. It is a fatal error for the number of data points to be less than the number of coefficients.

An example The following example shows how you could use these commands to fit a data set to the function `y = a*sin(x/10) + b*cos(x/20)`. The x values vary from 0 to 49 in steps of 1. The function `MakeFunc()` calculates the trial data set, to which we add random noise. We do not supply an array of sigma values for each data point; instead we give all points a value of 1.0, which means that `FitLinear()` returns the sum of squares of the errors between the fitted curve and the raw data values. If you run this example, you will notice that the returned value is slightly less than the sum of squares of the added errors.

```
const noise := 0.01;       ' controls how much noise we add
const NCOEF% := 2;         ' number of coefficients
const NDATA%:=50;          ' number of data values
var data[NDATA%];          ' space for our function
var x[NCOEF%][NDATA%];     ' array of function information
var err := 0.0;            ' the sum of squares of error we add

' Generate raw data. Fit y = a*sin(x/10)+b*cos(x/20)
var coef[NCOEF%], i%, r;   ' coefficients, index, random noise
coef[0]:=1.0; coef[1]:=2;  ' set coefficients for generated data
MakeFunc(data[], coef[], x[][]);

' Now add noise to the raw data values in data[]
for i%:=0 to NDATA%-1 do
   r := (rand()-0.5)*noise;' the noise to add
   data[i%] += r;          ' add noise to the data
   err := err + r*r;       ' accumulate sum of squared noise
   next;

var covar[NCOEF%][NCOEF%]; ' covariance array
var sig2, a[NCOEF%];       ' sigma, fitted coefficients
var rel[NCOEF%];           ' array for "relevance" values
sig2 := FitLinear(a[], data[], x[][], 1, covar[][], rel[]);
```

```
Message("sig^2=%g, err=%g\ncoefs=%g\nrel=%g",sig2,err,a[],rel[]);
halt;

'y[]  is the output array (x values are 0, 1, 2...)
'a[]  is the array of coefficients
' y = a*sin(x/10)+b*cos(x/20)
proc MakeFunc(y[], a[], x[][])
var nd%,v;                    ' coefficient index, work space
for nd% := 0 to NDATA%-1 do
       v := Sin(nd% / 10.0);    ' first funcion
       x[0][nd%] := v;          ' save the value;
       y[nd%] := a[0] * v;      ' start to build the result
       v := Cos(nd%/20.0);      ' second function
       x[1][nd%] := v;          ' save it
       y[nd%] += a[1]*v;        ' full result
   next;
end;
```

FitNLUser()

This command will use a non-linear fitting algorithm to fit a user-defined function to a set of data points. The function to be fitted must be of the form $y = f(x, \mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2\ldots)$ where the $\mathbf{a}_i$ are constants to be determined. You must be able to calculate the differential of the function f with respect to each of the coefficients. You can optionally supply an array to weight each data point. The commands to implement this are:

Set up the problem

The first command sets the user-defined function, the number of coefficients you want to fit, the number of data points and optionally, you can set the weight to give each data point. You must call this function before you call any of the others.

```
func FitNLUser(User(ind%, a[], dyda[]), nCoef%, nData%{, s[]|s});
```

`User()` A user-defined function which is called by the fitting routine. The function is passed the current values of the coefficients and returns the error between the function and the data point identified by `ind%` and the differentials of the function with respect to each of the coefficients at that point. The return value should be the y data value at the index minus the calculated value of the function at the x value, using the coefficients passed in.

`ind%` The index into the data points at which the error and differentials are to be evaluated. If there are `n` data points, `ind%` will run from 0 to `n`-1. You can rely on the function being called with the same coefficients as `ind%` increments from 0 to `n`-1, which may be useful if you have complex functions of the coefficients to evaluate.

`a` An array of length `nCoef%` holding the current values of the coefficients. The coefficients are refreshed for each call to the user-defined function, so it is not an error to change them; however this is usually not done.

`dyda` An array of length `nCoef%` which your function should fill in with the values of the partial differential of the function with respect to each of the coefficients. For example, if you were fitting $y = \mathbf{a_0}$`*exp(-`$\mathbf{a_1}$`*x)` then set `dyda[0]` = $\delta y/\delta\mathbf{a_0}$ = `exp(-`$\mathbf{a_1}$`*x)` and `dyda[1]` = $\delta y/\delta\mathbf{a_1}$ = $-\mathbf{a_0}$ `*`$\mathbf{a_1}$`*exp(-`$\mathbf{a_1}$`*x)`.

`nCoef%` The number of coefficients to fit in the range 1 to 10.

`nData%` The number of data points you will be fitting. If `s[]` is provided as an array, the value of `nData%` used is the smaller of `nData%` and the length of the `s[]` array. It is a fatal error for the number of data points used to be less than `nCoef%`.

`s` This argument is optional. It is either an array of weights to be given to each data point in the fit or a single weight to apply to all data points. If this value is the expected standard deviation of the y value of the data points, then the error value returned is the chi-squared value and the fit is a chi-squared fit. If this value is proportional to the expected error at the data point, then the fit is still a chi-squared fit, and the error returned is proportional to the chi-squared value. If you omit this argument, the fit is a least-squares error fit, and the error value returned is the sum of the squares of the errors in the y values.

Returns The `FitNLUser()` function returns 0. There is no other return value as all errors stop the script.

Unlike the other fitting routines, you will notice that the x and y data values are not passed into the command. Instead, the user-defined function is passed an index to the data values. It is assumed that the data is accessible by the user function.

Due to restrictions in the implementation of the script language, you cannot debug through the user-defined function. If you set a break point in it, or attempt to step into it, you will get errors. We recommend that you check the returned values from the user-defined function by calling it from your own script code.

Set coefficient values and ranges

This variant of the function sets the initial value of each coefficient and optionally sets the range of allowed values. You can call this function at any time after the setup call and before the iterate call (below) has returned 0 indicating that the fit is completed.

In this type of fitting it is very important that you give reasonable starting values for the coefficients. If you do not give starting values, the command will set them all to zero, which is unlikely to be correct.

```
func FitNLUser(coef%, val{, lo, hi});
```

`coef%` The coefficient to set. The first coefficient is number 0, the last one is `nCoef%-1`.

`val` The initial value to assign to the coefficient. If you have set low and high limits, and the value is outside these limits, it is set to the nearer limit.

`lo,hi` If present, these two values set the acceptable range of values for this coefficient. If omitted, or if both values are set to the same value, there is no limit. The value of the coefficient is tested against these limits after each iterative step, and if it exceeds a limit, it is set to that limit.

Iterate to a solution

Once you have set up the problem and given initial values to your coefficients, you can start the iteration process that will move the coefficients from their starting values to new values that minimise the error (optionally scaled by the `s` argument).

```
func FitNLUser(a[], &err{, maxI%{, &iTer%{, covar[][]}}});
```

`a[]` An array of size at least `nCoef%` that is returned holding the current set of coefficient values.

`err` A real variable returned as the sum over the data points of $(y_{\texttt{x[i]}}-\texttt{y[i]})^2/\texttt{s[i]}^2$ if `s[]` is used or holding the sum of $(y_{\texttt{x[i]}}-\texttt{y[i]})^2$ if `s[]` is not used, where $y_{\texttt{x[i]}}$ is the value predicted from the coefficients at the x value `x[i]`.

`maxI%` This is the maximum number of fitting iterations to do before returning from the function. If you omit this value, the function sets 1000. You can set any value from 1 to 10000. If you set more than 10000, the number is limited to 10000.

`iTer%` An optional integer variable that is returned holding the number of iterations done before the function returned.

`covar` An optional two dimensional array of size at least `[nCoef%][nCoef%]` that is returned holding the covariance matrix when the fit is complete. It is not changed unless the function return value is 0 or -1.

Returns This call returns 1 if the number of iterations has been completed, but the fitting process has not yet converged, 0 if the fitting process has converged, and a negative number if the fitting process is going nowhere.

Remember that even when a minimum is found, there is no guarantee that it is *the* minimum. It is the best minimum that this algorithm can find given the starting point.

Select coefficients to fit

Normally the command will fit all the coefficients, but you can use this command variant to select the coefficients to fit. You can use this command at any time after you have set the problem until the iteration variant returns 0 or -1.

```
func FitNLUser(fit%[]);
```

`fit%[]` An integer array of length at least `nCoef%`. Each element specifies if the corresponding coefficient is to be fitted (`fit%[i] := 1`) or held constant (`fit%[i] := 0`). If all elements are 0, all arguments are fitted.

The effect of this command persists until either the iteration variant returns a value less than 1, or you set up a new problem, or you call this variant again.

An example

The following is an example of using this set of commands to fit the user-defined function `y = a * exp(-b*x)`. In this example we generate some test data and add to it a random error. There are two coefficients to be fitted (**`a`** and **`b`**).

```
const NDATA%:=100;          ' number of data points
const NCOEF% := 2;          ' number of coefficients
var x[NDATA%],y[NDATA%],i%;

' generate data: a:=1, b:=0.05 and add some noise
for i% := 0 to NDATA%-1 do
   x[i%] := i%;
   y[i%] := exp(-0.05*i%)+(rand()-0.5)*0.01;
   next;

' Now link in user function and set coefficient ranges
FitNLUser(UserFnc, NCOEF%, NDATA%);
FitNLUser(0, 0.5, 0.01, 2);   'Set range of amplitude
FitNLUser(1, 0.01, 0.001, 1); 'Set range of exponent

var coefs[NCOEF%], err, iter%;
i% := FitNLUser(coefs[], err, 100, iter%);
Message("fit=%d, Err=%g, iter=%d, coefs=%g", i%, err, iter%,
coefs[]);
halt;

' The user-defined function: y = a * exp(-b*x);
' dy/da = exp(-b*x)
' dy/db = -x * a * exp(-b*x)
func UserFnc(ind%, a[], dyda[])
var xi,yi,r;
xi := x[ind%];             ' local copy of x value
yi := y[ind%];             ' local copy of y value
dyda[0] := exp(-a[1]*xi); ' differential of y with respect to a
r := dyda[0] * a[0];       ' intermediate value
dyda[1] := -xi * r;        ' differential of y with respect to b
return yi-r;
end
```

FitPoly()

This command fits $y = \mathbf{a}_0 + \mathbf{a}_1 x + \mathbf{a}_2 x^2 + \mathbf{a}_3 x^3$ … to a set of (x,y) data points. If you can provide error estimates for each y value, you can use the covariance output from this command to provide confidence limits on the calculated coefficients and you can use the returned χ^2 value to test if the model is likely to fit the data. The command is:

```
func FitPoly(coef[], y[], x[]{, s[]|s{, covar[][]}});
```

`coef[]` A real array which sets the number of coefficients to fit and which returns the best fit set of coefficients. The array must be between 2 and 10 elements long. The coefficient $\mathbf{a}_0$ is returned in `coef[0]`, $\mathbf{a}_1$ in `coef[1]` and so on.

`y[]` A real array of y values. The smaller of the sizes of the `x[]` and `y[]` arrays (and `s[]` array, if provided), sets the number of data points. It is a fatal error for the number of data points to be less than the number of coefficients.

`x[]` A real array of x values.

`s` This is an optional argument. It is either a real array holding the standard deviations of each of the `y[]` data points, or it is a real value holding the standard deviation of all of the data points. If the argument is omitted or set to zero, a least squares error fit is performed, otherwise a chi-squared fit is done.

`covar` An optional two dimensional array of size at least [*nc*][*nc*] (*nc* is the number of coefficients fitted) that is returned holding the covariance matrix.

Returns The function returns the chi-squared value for the fit if `s[]` or `s` is given (and non-zero), or the sum of squares of the errors between the data points and the best fit line if `s` is omitted or is zero.

An example

The following example generates a set of test data, adds random noise to it, then fits a polynomial to the data.

```
const NCOEF% := 5;          ' number of coefficients
const NDATA%:=50;           ' number of data points
var y[NDATA%];              ' space for our function
var x[NDATA%];              ' x co-ordinates
const noise := 1;           ' noise to add
var err := 0.0;             ' will be sum of squares of added noise
var cf[NCOEF%], i%, r;
cf[0]:=1.0; cf[1]:=-80; cf[2]:=-2.0; cf[3]:=0.5; cf[4]:=-0.009;
MakePoly(cf[],x[],y[]);     ' generate ideal data as polynomial
for i%:=0 to NDATA%-1 do    ' now add some noise to it
   r := (rand()-0.5)*noise;
   y[i%] += r;              ' add noise to the data
   err += r*r;              ' sum of squares of added noise
   next;
var sig2, a[NCOEF%];        ' a[] will be the fitted coefficients
sig2 := FitPoly(a[], y[], x[]);
Message("sig2=%g, noise=%g\nfitted=%8.4f\nideal =%8.4f",
                                         sig2, err, a[], cf[]);
halt;
```

```
'a[]  input array of coefficients
'x[]  output x co-ordinates, y[] output data values
proc MakePoly(a[], x[], y[])
var i%,j%,xv,s;
for i% := 0 to Len(y[])-1 do
   s := 0.0;
   xv := 1;
   for j% := 0 to NCOEF%-1 do
      s += a[j%]*xv;
      xv *= i%;
      next;
   y[i%] := s;
       x[i%] := i%;
   next;
end;
```

FitSin()

This command will fit multiple sinusoids to arrays of x,y data points, with an optional weight for each point. Fitting is by an iterative method. The input data is fitted to the equation:

$y = \mathbf{a}_0 \sin(\mathbf{a}_1 x + \mathbf{a}_2) + \mathbf{a}_3 \sin(\mathbf{a}_4 x + \mathbf{a}_5) + \ldots$

The fitted parameters (coefficients) are the $\mathbf{a}_i$. The angles are evaluated in radians. You can fit up to 3 sinusoids. Although the function is given in terms of sine functions, you can easily convert to cosines by subtracting $\pi/2$ from the phase angle ($\mathbf{a}_2$, $\mathbf{a}_5$, $\mathbf{a}_8$) after the fit. The commands to implement this are:

Set up the problem

The first command sets the number of sinusoids to fit and the data set to be fitted. You must call this function before you call any of the others.

```
func FitSin(nCoef%, y[], x[]{, s[]|s});
```

`nCoef%` The number of coefficients to fit. The only legal values are 3, 6 and 9 for one, two and three sinusoids.

`y[]` An array of y data values. The length of the array must be at least `nCoef%`.

`x[]` An array of x data values. The length of the array must be at least `nCoef%`.

`s` An optional argument which is either an array with one value for each y data point or a single value for all data points. If the value is the standard deviation, then the error value returned when you iterate to find the best fit is the chi-squared value and the fit is a chi-squared fit.

If this value is proportional to the error in the y values, then the fit is still a chi-squared fit, and the error returned is proportional to the chi-squared value. If you omit this array, the fit is a least squares error fit, and the error value returned is the sum of the squares of the errors in the y values.

Returns The function returns 0. There is no other return value as all errors stop the script.

The number of data points is set by the smallest of the sizes of the `y[]`, `x[]` or `s[]` (if present) arrays. The number of data points must be at least the number of coefficients. If it is not you will get a fatal error, so check this before calling the function.

Set coefficient values and ranges

This variant of the function sets the initial value of each coefficient and optionally sets the range of allowed values. You can call this function at any time after the setup call and before the iterate call (below) has returned 0, indicating that the fit is completed.

In this type of fitting it is very important that you give reasonable starting values for the coefficients. In particular, when fitting multiple sinusoids you will usually either know, or have a good idea of, the frequencies. You should limit the range of each frequency so that they cannot overlap. If you can do this, the fit will proceed quickly. If you do not give starting values, the command will make a simplistic guess at the fitting values. As we expect that you know more about the "right" answer than the command does, we suggest that you set the values you want.

```
func FitSin(coef%, val{, lo, hi});
```

`coef%` The coefficient to set. The first coefficient is number 0, the last one is `nCoef%`-1.

`val` The initial value to assign to the coefficient. If you have set low and high limits, and the value is outside these limits, it is set to the nearer limit.

`lo,hi` If present, these two values set the acceptable range of values for this coefficient. If omitted, or if both values are set to the same value, there is no limit. The value of the coefficient is tested against these limits after each iterative step, and if it exceeds a limit, it is set to that limit.

Iterate to a solution

Once you have set up the problem and given initial values to your coefficients, you can start the iteration process that will move the coefficients from their starting values to new values that minimise the error (optionally scaled by the `s` argument).

```
func FitSin(a[], &err{, maxI%{, &iTer%{, covar[][]}}});
```

`a[]` An array of size at least `nCoef%` that is returned holding the current set of coefficient values. The first amplitude is in `a[0]`, the first frequency in `a[1]`, the first phase angle in `a[2]`, the second amplitude in `a[3]` and so on.

`err` A real variable returned as the sum over the data points of $(y_{x[i]} - y[i])^2 / s[i]^2$ if `s[]` is used or holding the sum of $(y_{x[i]} - y[i])^2$ if `s[]` is not used, where $y_{x[i]}$ is the value predicted from the coefficients at the x value `x[i]`.

`maxI%` This is the maximum number of fitting iterations to do before returning from the function. If you omit this value, the function sets 1000. You can set any value from 1 to 10000. If you set more than 10000, the number is limited to 10000.

`iTer%` An optional integer variable that is returned holding the number of iterations done before the function returned.

`covar` An optional two dimensional array of size at least `[nCoef%][nCoef%]` that is returned holding the covariance matrix when the fit is complete. It is not changed unless the function return value is 0 or -1.

Returns This call returns 1 if the number of iterations has been completed but the fitting process has not yet converged, 0 if the fitting process has converged, and a negative number if the fitting process is going nowhere.

Even when a minimum is found, there is no guarantee that this is *the* minimum, only that it is the best minimum that this algorithm can find given the starting point.

Select coefficients to fit

Sometimes you may wish to hold some coefficients fixed while you fit others. Normally the command will fit all the coefficients, but you can use this command variant to select the coefficients to fit. You can use this command at any time after you have set the problem until the iteration variant returns 0 or -1.

```
func FitSin(fit%[]);
```

`fit%[]` An integer array of length at least `nCoef%`. Each element specifies if the corresponding coefficient is to be fitted (`fit%[i] := 1`) or held constant (`fit%[i] := 0`). If all elements are 0, then all arguments are fitted.

The effect of this command persists until either the iteration variant returns a value less than 1, or you set up a new problem, or you call this variant again. For a sinusoidal fit it is likely that you will know the frequency to fit, so you may well hold this constant.

An example The following is a template for using this set of commands (assuming you don't want to fit the frequency, which we assume you know).

```
const nData%:=50;          'set number of data elements
var x[nData%], y[nData%]; 'space for our arrays
var s[nData%];             'space for sigma of each point
var fit%[3];               'we want to hold the frequency
var coef[4];               'space for coefficients
var err;                   'will hold error squared
...                        'in here goes code to get the data
FitSin(3, y[], x[], s[]); 'fit one sinusoid

'Note that we let the phase take any value
FitSin(0, 1.0, 0.2, 4);   'set amplitude and limit range
FitSin(1, .02, .01, .03); 'set frequency
FitSin(2, 0., 0.3, 1.9);  'set width and limit range

'Now we say that we don't want to fit the frequency
ArrConst(fit%[],1);       'set all elements to 1
fit%[1] := 0;             'but not element 1 (=frequency)
FitSin(fit%[]);           'so the frequency is fixed

repeat
   DrawMyData(coefs[], x[], y[]);  'Some function to show progress
until FitSin(coefs[], err, 1) < 1;

DrawMyData(coefs[], x[], y[]);     'Show the final state
```

FitValue()

This function returns the value at a particular x axis point of the fitted function set by the last `FitData()` command.

```
Func FitValue(x);
```

x The x axis value at which to evaluate the current fit. You should be aware that some of the fitting fuctions can overflow the floating point range if you ask for x values beyond the fitted range of the function.

Returns The value of the fitted function at x. If the result is out of floating point range, the function may return a floating point infinity or a NaN (Not a Number) value or a 0. If there is no fit, the result is always 0.

See also: `FitCoef()`, `FitData()`, `FitExp()`, `ChanFitValue()`

Floor()

Returns the next lower integral number of the real number or array. `Floor(4.7)` is `4.0`, `Floor(4)` is `4`. `Floor(-4.7)` is `-5`.

```
Func Floor(x|x[] {[]…});
```

`x` A real number or a real array.

Returns When the argument is an array, the function replaces the array with the next lower integral number of all the points and returns either a negative error code or 0 if all was well.
When the argument is not an array the next lower integral number.

See also: `Abs()`, `ATan()`, `Ceil()`, `Cos()`, `Exp()`, `Frac()`, `Ln()`, `Log()`, `Max()`, `Min()`, `Pow()`, `Rand()`, `Round()`, `Sin()`, `Sqrt()`, `Tan()`, `Trunc()`

FocusHandle()

This function returns the view handle of the script-controllable window with the input focus (the active window). Unlike FrontView(), it can return any type of window, for example the toolbar.

```
Func FocusHandle();
```

Returns The handle of a window that the script can manipulate, or 0 if the focus is not in such a window.

See also: `FrontView()`

FontGet()

This function gets the name of the font, and its characteristics for the current view.

```
Func FontGet(&name$, &size, &style%);
```

`name$` This string variable is returned holding the name of the font.

`size` The real number variable is returned holding the point size of the font.

`style%` This integer variable is returned holding the style:
- 0 Normal text.
- 1 Italic text.
- 2 Bold text.
- 3 Bold and italic.

Returns The function returns 0 if all was well or a negative error code. If an error occurs, the variables are not changed.

See also: `FontSet()`

FontSet()

This function sets the font for the current view. This does not cause an immediate redraw; use the `Draw()` command to force one.

```
Func FontSet(name$|code%, size, style%);
```

`name$` This is a string holding the name of the font to use. Alternatively, you can specify a font code:

`code%` This is an alternative (system independent) method of specifying a font. We recognise these font codes:
- 0 The standard system font, whatever that might be.
- 1 A non-proportionally spaced font, usually Courier-like.
- 2 A proportionally spaced non-serifed font, such as Helvetica or Arial.
- 3 A proportionally spaced serifed font, such as Times Roman.
- 4 A symbol font.
- 5 A decorative font, such as Zapf-Dingbats or TrueType Wingdings.

`size` The point size required. Each system will set a minimum and maximum point size. Values outside this range are limited to the extreme values.

`style%` This integer value sets the type style (if supported on your system):
- 0 Normal text.
- 1 Italic text.
- 2 Bold text.
- 3 Bold and italic.

Returns The function returns 0 if the font change succeeded, or a negative error code.

See also: `FontGet()`

Frac()

Returns the fractional part of a real number or replaces an array of reals with its fractional parts.

```
Func Frac(x|x[]{[]...});
```

`x` A real number or an array of reals.

Returns For arrays, it returns 0 or a negative error code. If `x` is not an array it returns a real number equal to `x-Trunc(x)`. E.g. `Frac(4.7)` is 0.7, `Frac(-4.7)` is -0.7.

See also: `Trunc()`, `Round()`

Frame()

A function to get or set the current frame in a data view.

```
Func Frame({frame%});
```

`frame%` If this is present and in range, the current frame changes to the new number.

Returns The frame number for the view at the time of the call.

See also: `FrameComment$()`, `FrameCount()`, `FrameFlag()`, `FrameTag()`, `FrameUserVar()`

FrameAbsStart()

Obtains the absolute start time for the current frame in a data view. The absolute start time for a frame is the time for time zero in a frame relative to the time at which sampling was started.

```
Func FrameAbsStart();
```

Returns The absolute start time, in seconds, of the current frame in the current data view.

See also: `Frame()`, `SampleStart()`

FrameComment$()

Gets or sets the comment with the current frame. This is a string of up to 72 characters that is stored with each frame.

```
Func FrameComment$({c$});
```

`c$` If this is present it provides a new comment to store with the frame.

Returns The frame comment at the time of the call.

See also: `FileComment$()`, `FrameState()`, `FrameTag()`, `FrameUserVar()`

FrameCount()

Obtains the number of frames in a document.

```
Func FrameCount();
```

Returns Number of frames (sweeps) in the file or memory view.

See also: `Frame()`, `Sweeps()`

FrameFlag()

This command turns a frame flag on or off or retrieves the current setting of a flag from the specified frame.

```
Func FrameFlag(frm%|frm$|frm%[], flag%{, set%});
```

`frm%` Frame number or a negative code as follows:

- -1 All frames in the file.
- -2 The current frame.
- -3 Only tagged frames.
- -6 Only untagged frames.

`frm$` A frame specification string. This option specifies a list of frames using a string such as “1..32,40,50”.

`frm%[]` An array of frame numbers. This option provides a list of frame numbers. The first element holds the number of frames in the list.

`flag%` The flag number (1..32). For CFS files not created by Signal, only flags 1 to 4 and flag 16 are present, though the other flags can be used while the frame is held in memory.

`set%` If non-zero, this sets flag number `flag%` on in the current frame. If zero, it turns the flag off. In a file not created by Signal only flags 1,2,3,4 or 16 can be permanently changed.

Returns 1 if the flag number `flag%` is set in the last frame specified when the function was called or zero if not. Returns -1 if no frames are found to match the specification.

See also: `FrameComment$(), FrameState(), FrameTag(), FrameUserVar()`

FrameGetIntVar()

This function reads a CFS frame variable of integer type from the current frame.

```
Func FrameGetIntVar(name$ {&nVar%{, &units$, {nType%}}});
```

`name$` The name of the variable to look for. This string is not case sensitive but every character including spaces must match exactly.

`nVar%` If present this returns the variable number, -1 if not found, or a negative error code.

`units$` If present this returns the units for the variable.

`nType%` If present this returns a code for the CFS type of an integer variable:

0: INT1, 1: WRD1, 2: INT2, 3: WRD2, 4: INT4:

Returns The function returns the value of the variable if the operation was a success, otherwise zero.

See also: `FileGetIntVar(), FileGetRealVar(), FileGetStrVar$(), FileVarCount(), FileVarInfo(), FrameGetRealVar(), FrameGetStrVar$(), FrameVarCount(),FrameVarInfo()`

FrameGetRealVar()

This function reads a CFS frame variable of real type from the current frame.

```
Func FrameGetRealVar(name$ {&nVar%{, &units$}});
```

`name$` The name of the variable to look for. This string is not case sensitive but every character including spaces must match exactly.

`nVar%` If present this returns the variable number, -1 if not found, or a negative error code.

`units$` If present this returns the units for the variable.

Returns The function returns the value of the variable if the operation was a success, otherwise zero.

See also: `FileGetIntVar(), FileGetRealVar(), FileGetStrVar$(), FileVarCount(), FileVarInfo(), FrameGetIntVar(), FrameGetStrVar$(), FrameVarCount(), FrameVarInfo()`

FrameGetStrVar$()

This function reads a CFS frame variable of string type from the current frame.

```
Func FrameGetStrVar$(name$ {&nVar%{, &units$}});
```

`name$` The name of the variable to look for. This string is not case sensitive but every character including spaces must match exactly.

`nVar%` If present this returns the variable number, -1 if not found, or a negative error code.

`units$` If present this returns the units for the variable.

Returns The function returns the string contents of the variable if the operation was a success, otherwise an empty string.

See also: `FileGetIntVar(), FileGetRealVar(), FileGetStrVar$(), FileVarCount(), FileVarInfo(), FrameGetIntVar(), FrameGetRealVar(), FrameVarCount(), FrameVarInfo()`

FrameList()

This function generates an array of selected frame numbers from the current view.

```
Func FrameList(list%[], sFrm%{, eFrm%{,mode%}});
```

```
Func FrameList(list%[], frm$|frm%[]{,mode%});
```

`list%` An integer array to fill with frame numbers. The first element of the array, `list%[0]`, is set to the number of frames returned, and the remaining elements in the array are frame numbers. If the array is too short, enough frames are returned to fill the array.

`sFrm%` First frame to include. This option returns a range of frames. `sFrm%` can also be a negative code as follows:

-1 All frames in the file are included.
-2 The current frame.
-3 Frames must be tagged.
-6 Frames must be untagged.

`eFrm%` Last frame to include. If this is -1 the last frame number in the data file is used. This argument is ignored if `sFrm%` is a negative code.

`frm$` A frame specification string. This option specifies a list of frames using a string such as “1..32,40,50”.

`frm%[]` An array of frame numbers. This option provides a list of frame numbers. The first element holds the number of frames in the list.

`mode%` If `mode%` is present it is used to supply an additional criterion for including each frame in the range, list or specification. If `mode%` is absent all frames are included. The modes are:

0-n Frames must have a state matching the value of `mode%`.
-1 All frames in the range list are included.
-2 Only the current frame in the view is included.
-3 Frames must be tagged.
-6 Frames must be untagged.

Returns The number of frames that would be returned if the array was of unlimited length or 0 if the view is not a data view.

See also: `FrameState()`, `FrameTag()`, `FrameUserVar()`, `ExportFrameList()`, `ProcessFrames()`

FrameMean()

This command turns the frame mean flag on or off or gets the setting of the mean flag for the specified frame or frame type.

```
Func FrameMean(frm%|frm$|frm%[]{, on%});
```

`frm%` One frame number or a negative code as follows:

- -1 All frames in the file.
- -2 The current frame.
- -3 All tagged frames.
- -6 All untagged frames.

`frm$` A frame specification string. This option specifies a list of frames using a string such as “1..32,40,50”.

`frm%[]` An array of frame numbers. This option provides a list of frame numbers. The first element holds the number of frames in the list.

`on%` A value of 1 marks the frame as a mean, zero marks it as a total.

Returns 1 if the last frame specified was a mean when the function was called or zero if not. Returns -1 if no frames are found to match the specification.

See also: `FrameComment$()`, `FrameFlag()`, `FrameState()`, `FrameUserVar()`, `Sweeps()`

FrameSave()

This command saves changed frame data in a file view back into the file, bypassing the usual interactive process controlled by the preferences dialog. It can also be used to discard changes to ensure that the user is not promped to save them. This command can only be used on frames already present on disk; appended frames and memory view frames will be saved as part of `FileSave` or `FileClose`.

```
Func FrameSave({no%});
```

`no%` If present and non-zero, this causes changed data to be discarded by marking the data as unchanged. If the parameter is not present or set to zero the function causes the changed data to be written back to the disk file.

Returns Zero or a negative error code.

See also: `FileExportAs()`, `FileSave()`, `FileClose()`, `Frame()`

FrameState()

This command sets or gets the value of the state variable for the specified frame or frames.

```
Func FrameState(frm%|frm$|frm%[]{, new%});
```

`frm%` One frame number or a negative code as follows:

-1 All frames in the file
-2 The current frame
-3 All tagged frames
-6 All untagged frames

`frm$` A frame specification string. This option specifies a list of frames using a string such as “1..32,40,50”.

`frm%[]` An array of frame numbers. This option provides a list of frame numbers. The first element holds the number of frames in the list.

`new%` If present this sets the state stored with each frame specified. For values above 9 this is only effective in a file created by Signal.

Returns The state of the last frame specified when the function was called. Returns -1 if no frames are found to match the specification.

See also: `FrameComment$(), FrameFlag(), FrameTag(), FrameUserVar()`

FrameTag()

This command turns the tag on or off or gets the setting of the tag for the specified frame or frame type.

```
Func FrameTag(frm%|frm$|frm%[]{, on%});
```

`frm%` One frame number or a negative code as follows:

-1 All frames in the file.
-2 The current frame
-3 All tagged frames
-6 All untagged frames

`frm$` A frame specification string. This option specifies a list of frames using a string such as “1..32,40,50”.

`frm%[]` An array of frame numbers. This option provides a list of frame numbers. The first element holds the number of frames in the list.

`on%` A value of 1 tags the frame, zero untags it.

Returns 1 if the last frame specified was tagged when the function was called or zero if not. Returns -1 if no frames are found to match the specification.

See also: `FrameComment$(), FrameFlag(), FrameState(), FrameUserVar()`

FrameUserVar()

This command gets or sets one of the user variables stored with the current frame.

```
Func FrameUserVar(n%{,val});
```

`n%` The user variable number, between 1 and 16.

`val` If present this is the new value for user variable number `n`.

Returns The value of frame user variable number `n` before the call.

See also: `FrameComment$(), FrameFlag(), FrameState(), FrameTag()`

FrameVarCount()

This function counts CFS frame variables in the data file.

```
Func FrameVarCount();
```

Returns The number of frame variables in the data file associated with this view.

See also: `FileGetIntVar()`, `FileGetRealVar()`, `FileGetStrVar$()`, `FileVarCount()`, `FileVarInfo()`, `FrameGetIntVar()`, `FrameGetRealVar()`, `FrameGetStrVar$()`, `FrameVarInfo()`

FrameVarInfo()

This function reads a description of a CFS frame (DS) variable.

```
Func FrameVarInfo(nVar%, &name$);
```

`nVar%` This is the variable number for which information is required. The first frame variable is number zero.

`name$` This is returned holding the name of the variable, which can be used in the commands for reading the frame variable values.

Returns The function returns the type of the variable or -1 if the variable was not found or is of unknown type. The variable type codes are as follows:

0 An integer variable which can be read using `FrameGetIntVar()`.
1 A floating point variable which can be read using `FrameGetRealVar()`.
2 A string variable which can be read using `FrameGetStrVar$()`.

See also: `FileGetIntVar()`, `FileGetRealVar()`, `FileGetStrVar$()`, `FileVarCount()`, `FileVarInfo()`, `FrameGetIntVar()`, `FrameGetRealVar()`, `FrameGetStrVar$()`, `FrameVarCount()`

FrontView()

This command is used to set the view that is nearest to the top and makes it the current view. It is the view that would have the focus if all dialogs were removed. You can use this to find out the front view, or to set it. When a view becomes the front view, it is moved to the front unless it is already there. If an invisible or iconised view is made the front view, the view is made visible automatically (equivalent to `WindowVisible(1)`). Care should be taken if using this function in an idle routine for a toolbar, as calling it repeatedly will prevent the toolbar buttons from being pressed!

```
Func FrontView( {vh%} );
```

`vh%` Either 0 or omitted to return the front view handle, the handle of the view to be set, or -n, meaning the n^{th} duplicate of the data view associated with the current view.

Returns 0 if there are no visible views, -1 if the view handle passed is not a valid view handle, otherwise it returns the view handle of the view that was at the front.

See also: `View()`, `Window()`, `WindowVisible()`, `FocusHandle()`

GammaP()

This is the incomplete gamma function $P(a, x)$ and is defined mathematically as:

$$P(a, x) = 1/\Gamma(a) \int_0^x e^{-t} t^{a-1} dt$$

$\Gamma(a)$ is the gamma function described under `LnGamma()`. From the incomplete gamma function is obtained the error function, the cumulative Poisson probability function and the Chi-squared probability function.

The error function $erf(x) = 2/\sqrt{\pi} \int_0^x e^{-t^2} dt =$ `GammaP(0.5, x*x)`

The cumulative Poisson probability function relates to a Poisson process of random events and is the probability that, given an expected number of events `r` in a given time period, the actual number was greater than or equal to `n`. This turns out to be `GammaP(n,r)`. Also, the probability that there are less than n events is `GammaQ(n,r)` (described below).

The Chi-squared probability function is useful where we are fitting a model to data. Given a fitting function that fits the data with `n` degrees of freedom (if you have `nData` data points and `nCoef` coefficients you usually have `nData-nCoef` degrees of freedom), and given that the errors in the data points are normally distributed, the probability of a Chi-squared value less than `chisq` is `GammaP(n/2, chisq/2)`. Similarly, the probability of a `chisq` value at least as large as `chisq` is `GammaQ(n/2, chisq/2)`. So, if you know the chi-squared value from a fitting exercise, you can ask "What is the probability of getting this value (or a greater one) given that my model fits the data?" If the probability is very small, it is likely that your model does not fit the data, or your fit has not converged to the correct solution.

```
Func GammaP(a, x);
```

`a` This must be positive, it is a fatal error if it is not.

`x` This must be positive, it is a fatal error if it is not.

Returns The incomplete Gamma function.

GammaQ()

The complement of GammaP(); `GammaQ(a,x)` is `1.0-GammaP(a,x)`.

```
Func GammaQ(a, x);
```

`a` This must be positive, it is a fatal error if it is not.

`x` This must be positive, it is a fatal error if it is not.

Returns The complement of the incomplete Gamma function.

Grid()

This function turns the background grid on and off for the current data or XY view. It also returns the state of the grid.

```
Func Grid({on%});
```

`on%` Optional argument that sets the grid state. If this is omitted, no change is made.

- 0 Turn the grid off.
- 1 Turn the grid on.

Returns The state of the grid at the time of the call, or a negative error code. Changes made by this function do not cause an immediate redraw.

Gutter()

The gutter is the area on the left of a text-based window where bookmarks and script break points appear. This function returns and optionally sets the gutter visible state. If you set a large font size you may wish to hide the gutter.

```
Func Gutter({show%});
```

`show%` Optional, sets the gutter state. 0 = hide, 1 = show, -1 or omitted for no change.

Returns The gutter state at the time of the call: 0 = hidden, 1=visible.

HCursor()

This function returns the y axis position of a horizontal cursor, and optionally sets a new position. You can get and set positions of cursors attached to invisible channels or channels that have no y axis.

```
Func HCursor(num% {,where {,chan%}});
```

`num%` The cursor to use. It is an error to attempt this operation on an unknown cursor.

`where` If this parameter is given it sets the new y axis position of the cursor.

`chan%` If this parameter is given, it sets the channel number (1 to n).

Returns The function returns the y axis position of the cursor at the time of the call, or zero for a non-existent cursor.

See also: `Cursor()`, `HCursorChan()`, `HCursorDelete()`, `HCursorLabel()`, `HCursorLabelPos()`, `HCursorNew()`, `HCursorRenumber()`

HCursorChan()

This function returns the channel number that a particular horizontal cursor is currently attached to.

```
Func HCursorChan(num%);
```

`num%` The horizontal cursor number.

Returns It returns the channel number that the cursor is attached to, or 0 if this cursor is not attached to any channel or if the channel number is out of the allowed range.

See also: `HCursor()`, `HCursorDelete()`, `HCursorLabel()`, `HCursorLabelPos()`, `HCursorNew()`, `HCursorRenumber()`

HCursorDelete()

This deletes the designated horizontal cursor. It is not an error to delete an unknown cursor, it just has no effect.

```
Func HCursorDelete({num%});
```

`num%` The number of the cursor to delete, or -1 to delete all horizontal cursors. If this is omitted, the highest numbered cursor is deleted.

Returns The number of the deleted cursor or 0 if no cursor was deleted.

See also: `CursorDelete()`, `HCursor()`, `HCursorChan()`, `HCursorLabel()`, `HCursorLabelPos()`, `HCursorNew()`, `HCursorRenumber()`

HCursorExists()

This function tests if a given horizontal cursor exists at the time of the call.

```
Func HCursorExists(num%)
```

`num%` The cursor number, between 1 and 4.

Returns 1 if the cursor exists, 0 if it does not.

See also: `HCursor(), HCursorDelete(), CursorExists()`

HCursorLabel()

This command sets (or gets) the horizontal cursor label style for the current view.

```
Func HCursorLabel({style%{, num%{, form$}}})
```

`style%` The cursor style. Cursors can be annotated with a position or the cursor number or a user-defined style. The styles are: 0=Neither, 1=Position, 2=Number, 3=Both, 4=User-defined. Unknown styles cause no change.

`num%` The cursor number for style 4. 0 means all cursors, 1-4 for a single cursor.

`form$` The label string for style 4. The string has replaceable parameters `%p`, and `%n` for position and number. We also allow `%w.dp` where `w` and `d` are numbers that set the field width and decimal places. You cannot read back a label format string.

Returns The previous cursor style. If you omit `style%`, the style does not change.

See also: `CursorLabel(), HCursor(), HCursorChan(), HCursorDelete(), HCursorLabelPos(), HCursorNew(), HCursorRenumber()`

HCursorLabelPos()

This lets you set and read the position of the horizontal cursor label.

```
Func HCursorLabelPos(num% {,pos});
```

`num%` The cursor number, between 1 and 4. If the cursor does not exist the function does nothing and returns -1.

`pos` If present, the command sets the position. The position is a percentage of the distance from the left of the cursor at which to position the value. Out of range values are set to the appropriate limit.

Returns The cursor position before any change was made, or -1 if the cursor does not exist.

See also: `CursorLabelPos(), HCursor(), HCursorChan(), HCursorDelete(), HCursorLabel(), HCursorNew(), HCursorRenumber()`

HCursorNew()

This function creates a new horizontal cursor and assigns it to a channel. You can create up to 4 horizontal cursors, which can subsequently be moved to any channel.

```
Func HCursorNew(chan% {,where});
```

`chan%` A channel number (1 to n) for the new cursor. If the channel is hidden the cursor is hidden.

`where` An optional argument setting the cursor position. If this is omitted, the cursor is placed in the middle of the y axis or at zero if there is no y axis.

Returns It returns the horizontal cursor number or 0 if all cursors are in use.

See also: `CursorNew(), HCursor(), HCursorChan(), HCursorDelete(), HCursorLabel(), HCursorLabelPos(), HCursorRenumber()`

HCursorRenumber()

This command renumbers the cursors starting with channel 1. Cursors in each channel are numbered from bottom to top. There are no arguments.

```
Func HCursorRenumber();
```

Returns The number of cursors found in the view.

See also: `CursorRenumber()`, `HCursor()`, `HCursorChan()`, `HCursorDelete()`, `HCursorLabel()`, `HCursorLabelPos()`, `HCursorNew()`

Help()

Signal uses the standard Windows help system.

```
Func Help(topic%|topic$ {,file$});
```

`topic%` A numeric code for the help topic. These codes are assigned by the help system author. Code 0 changes the default help file to `file$`.

`topic$` A string holding a help topic keyword or phrase to look-up.

`file$` If this is omitted, or the string is empty, the standard Signal help file is used. If this holds a filename, this filename is used as the help file.

Returns 1 if the help topic was found, 0 if it was not, -1 if the help file was not found.

The Windows SDK has some help-authoring tools, and third-party tools are available.

IIR commands

The `IIRxxxx()` script commands make it easy for you to generate and apply IIR (Infinite Impulse Response) filters to data held in arrays of real numbers. The data values are assumed to be a sampled sequence, spaced at equal intervals. You can create digital filters that are modelled on Butterworth, Bessel, Chebyshev type 1 and Chebyshev type 2 highpass, lowpass, bandstop and bandpass filters. You can also create digital resonators and notch filters. The commands are:

`IIRBp()`	Bandpass filter	`IIRHp()`	Highpass filter	`IIRNotch()`	Notch filter
`IIRBs()`	Bandstop filter	`IIRLp()`	Lowpass filter	`IIRReson()`	Resonator

The algorithms used to create the filters are based on the `mkfilter` program, written by Tony Fisher of York University. The basic idea is to position the s-plane poles and zeros for a normalised low-pass filter of the desired characteristic and order, then to transform the filter to the desired type.

The theory of IIR filters is beyond the scope of this manual; a classic reference work is *Theory and Application of Digital Signal Processing* by Rabiner and Gold, published in 1975. The IIR filters generated by these commands can be modelled by:

$$y[\mathrm{n}] = \sum_{i=0}^{N} a_i\, x[\mathrm{n}-i]/G + \sum_{i=1}^{M} b_i\, y[\mathrm{n}-i]$$

where the x[n] are the sequence of input data values, the y[n] are the sequence of output values, the a_i and the b_i are the filter coefficients (some of which may be zero) and G is the filter gain. Although G could be incorporated into the a_i, for computational reasons we keep it separate. In the filters designed by the `IIRxxxx()` commands, N=M and is the *order* of the filter for lowpass and highpass designs and is twice the order for bandpass and bandstop designs. The order of these filters determines the sharpness of the filter cut-off: the higher the order, the sharper the cut-off.

IIR and FIR filters

When compared to FIR filters, IIR filters have advantages:

- They can generate much steeper edges and narrower notches for the same computational effort.
- The filters are causal, which means that the filter output is only affected by current and previous data. If you run a step change through FIR filters you typically see ringing before the step as well as after it.

However, they also have disadvantages:

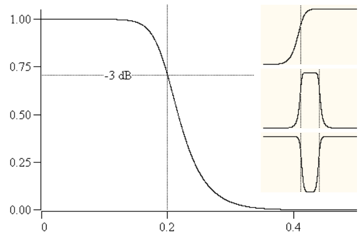
- IIR filters are prone to stability problems, FIR filters are unconditionally stable. All IIR filters generated by these commands should be stable, but high order band pass and band stop filters with narrow pass or stop bands may have problems.
- They impose a group delay on the data that varies with frequency. This means that they do not preserve the shape of a waveform, in particular, the positions of peaks and troughs will change.

You can remedy the group delay problem by running a filter forwards, then backwards, through the data. However, this makes the filter non-causal, removing one of the advantages of using an IIR filter. The commands allow you to check the impulse, step, frequency and phase response of the filters, and we recommend that you do so before using a generated filter for a critical purpose.

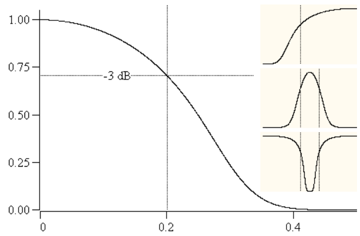
The lowpass, highpass, bandpass and bandstop filters generate digital filters modelled on four types of analogue filter: Butterworth, Bessel, Chebyshev type 1 and Chebyshev type 2. The resulting digital filters are not identical to the analogue filters as the mapping from the analogue to the digital domain distorts the frequency scale. In many cases, this improves the performance of the digital filter over the analogue counterpart.

Filter types You can generate notch and resonator filters plus lowpass, highpass, bandpass and bandstop filters modelled on Butterworth, Bessel and Chebyshev analogue filters. The examples for Butterworth, Bessel and Chebyshev filters show a fifth order lowpass filter with the edge set to 0.2 with inset examples of high-pass, bandpass and bandstop filters.

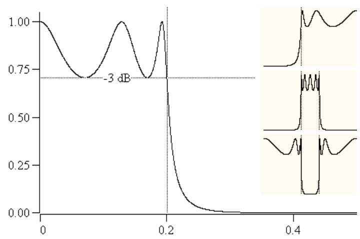
Butterworth These have a maximally flat pass band, but pay for this by not having the steepest possible transition between the pass band and the stop band. The example shows a low pass fifth order Butterworth filter with a cut-off frequency set to 0.2 of the sample rate. The x axis is frequency, the y axis is the filter gain. Both axes are linear.



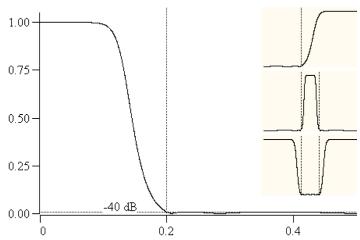
Bessel An analogue Bessel filter has the property that the group delay is maximally flat, which means that it tends to preserve the shape of a signal passed through it. This leads to filters with a gentle cut-off. When digitised, the constant group delay property is compromised; the higher the filter order, the worse the group delay. The example shows a fifth order low pass filter at 0.2 of the sample rate.



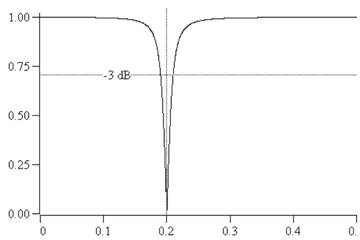
Chebyshev type 1 Filters of this type are based on Chebyshev polynomials and have the fastest transition between the pass band and the stop band for a given ripple in the pass band and no ripples in the stop band. In the example, the ripple has been set to 3 dB, to match the other examples, though you would normally choose less ripple than this.



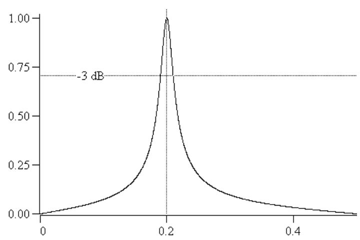
Chebyshev type 2 Filters of this type are defined by the start of the stop band and the stop band ripple. The filter has the fastest transition between the pass and stop bands given the stop band ripple and no ripple in the pass band. The example shows a fifth order filter with a 40 dB stop band ripple and with the stop band starting at 0.2 of the sample rate.



Notch Notch filters are defined by a centre frequency and a `q` factor. `q` is the width of the stop band at the –3 dB point divided by the centre frequency: the higher the `q`, the narrower the notch. Notch filters are often used to remove mains hum, but if you do this you will likely need to set notches at the first few odd harmonics of the mains frequency. The example has a centre frequency of 0.2 and a q of 10, so the width the –3 dB point is 0.02 of the sample rate.



Resonator A resonator is the inverse of a notch. It is defined in terms of a centre frequency and a `q` factor. `q` is the width of the pass band at the –3 dB point divided by the centre frequency: the higher the `q`, the narrower the resonance. Resonators are sometimes used as alternatives to a narrow bandpass filter. The example shows a centre frequency of 0.2 of the sample rate and a q of 10, so the width of the pass band at the –3 dB point is 0.02 of the sample rate.



Common command variants

There are six IIR commands. The commands are all independent and they each remember the last filter you created. They all support the following variants:

Get filter information

You can use a command variant of this form to return filter information:

```
Func IIRxxxx(get%{, arr[]{[]}});
```

`get%` The form of the command using this argument is used to read back information about the last filter created with this command. The argument can be:

1 `arr[]` is filled in with the impulse response of the filter. The return value is the magnitude of the largest value in `arr[]`. For example, if the impulse response ranged in values from –0.5 to 0.3, 0.5 would be returned.

2 `arr[]` is filled in with the step response of the filter. The return value is the magnitude of the largest value in `arr[]`.

3 `arr[][]` is a matrix with 2 columns and `r` rows. The frequency response is returned as complex numbers in the columns; column 0 holds the real part and column 1 the imaginary part. The first row corresponds to a frequency of 0; the final row corresponds to a frequency of half the sampling rate. The frequencies are spaced as `0.5/(r-1)`. The return value is the maximum magnitude of the returned frequency response (if you wish to normalise the response curve to a maximum of 1.0).

4 The same as 3 except that the results are returned as the amplitude response in column 0 and the phase response in column 1. If the real and imaginary parts of the response are `r` and `i`, column 0 holds `sqrt(r*r+i*i)` and column 1 holds `atan(i, r)`. The return value is the maximum returned amplitude response.

5 Returns the number of filter coefficients (`2*order%+1`) to apply to the filter input values and fills in `arr[]` with these values. These correspond to the a_i in the filter expression. However, we return the values in reverse order as this makes them easier to use as a dot product with old values.

6 Returns the number of filter coefficients (`2*order%+1`) to apply to the filter output values and fills in `arr[]` with these values. These correspond to the b_i in the filter expression. However, we return the values in reverse order to match the a_i. The final value is always –1.0 (corresponding to b_0, which is not used when implementing the filter).

7 Returns the filter gain G.

8 `arr[][]` is a matrix with 2 columns. The return value is the number of poles in the s-plane and the matrix is filled in with the poles as complex numbers with the real part in column 0 and the imaginary part in column 1.

9 The same as 8, but returning the s-plane zeros.

10 The same as 8, but returning the z-plane poles.

11 The same as 8, but returning the z-plane zeros.

Apply existing filter

This command applies the current filter to an array of equally spaced data. The variant with a single argument applies the filter forward through the array. With two arguments, and the second argument negative, the filter is applied backwards. Running the filter forwards introduces a phase shift into the output. Running the filter a second time, but backwards, cancels the phase shift at the expense of a non-causal filter.

```
Func IIRxxxx(data[]{, -1});
```

`data` An array of data to filter. If there is a second argument, it must be negative to request running the filter backwards.

See also: `FIRMake()`, `IIRBp()`, `IIRBs()`, `IIRHp()`, `IIRLp()`, `IIRNotch()`, `IIRReson()`

IIRBp()

This function creates and applies IIR (Infinite Impulse Response) band pass filters to arrays of data. You can run the filter forwards or backwards through the data.

```
Func IIRBp(data[]|0, lo, hi{, order%{, type%{, ripple}}});
Func IIRBp(data[]{, -1});
Func IIRBp(get%{, arr[]{[]}});
```

`data` An array of data to filter. If this is the only argument, or if there are 2 arguments and the second is negative, the last created band pass filter is used. Otherwise, the filter defined by the remaining arguments is used. Replace `data` with 0 to create a filter. Filters run forward through `data` unless there are 2 arguments and the second is negative, when the filter runs backwards.

`lo` The low corner frequency of the band stop filter. This is expressed as a fraction of the sample rate and is limited to the range 0.0002 to 0.4998. For Chebyshev type 2 filters, this is the point at which the attenuation reaches the `ripple` value, for all other filters this sets the –3 dB point.

`hi` The high corner frequency of the band pass filter. This is expressed as a fraction of the sampling rate and is limited to the range `lo`+0.0002 to 0.4998. For Chebyshev type 2 filters, this is the point at which the attenuation reaches the `ripple` value, for all other filters this sets the –3 dB point.

`order%` The order of the lowpass filter used as the basis of the design, in the range 1 to 10. If omitted, 2 is used. The order of the filter implemented is `order%*2`. High orders (`order%` > 7) and narrow bands may cause inaccuracy in the filter. Narrow means that `(hi-lo)/sqrt(lo*hi)` is less than 0.2, for example.

`type%` Set 0 for Butterworth, 1 for Bessel and 2 for Chebyshev type 1, 3 for Chebyshev type 2. The default value is 0 for a Butterworth filter.

`ripple` The desired pass band ripple in dB for Chebyshev type 1 filters (default 3 dB) or the desired minimum cut in the stop bands for Chebyshev type 2 filters (default 40 dB). The value here must be greater than 0.

`get%` The command variant with this argument returns information about the last filter you created with this command. See the discussion of IIR commands for details.

`arr` This is an option vector or matrix used to return information about the last filter you created with this command. See the discussion of IIR commands for details.

Returns All forms of the command return negative numbers for errors. The forms that apply or create filters return 0 for success. The other command forms have their return values included in the description of the `get%` argument or return 0.

See also:`FIRMake()`, `IIRBs()`, `IIRHp()`, `IIRLp()`, `IIRNotch()`, `IIRReson()`

IIRBs()

This function creates and applies IIR (Infinite Impulse Response) band stop filters to arrays of data. You can run the filter forwards or backwards through the data.

```
Func IIRBp(data[]|0, lo, hi{, order%{, type%{, ripple}}});
Func IIRBp(data[]{, -1});
Func IIRBp(get%{, arr[]{[]}});
```

`data` An array of data to filter. If this is the only argument or if there are 2 arguments and the second is negative, the last created filter is used. Otherwise, the filter defined by the remaining arguments is used. Replace `data` with 0 to create a filter. Filters run forward through `data` unless there are 2 arguments and the second is negative, when the filter runs backwards.

`lo` The low corner frequency of the band pass filter. This is expressed as a fraction of the sample rate and is limited to the range 0.0002 to 0.4998. For Chebyshev filters, this is the point at which the attenuation reaches the `ripple` value, for other filters this sets the –3 dB point.

`hi` The high corner frequency of the band pass filter. This is expressed as a fraction of the sampling rate and is limited to the range `lo`+0.0002 to 0.4998. For Chebyshev type 2 filters, this is the point at which the attenuation reaches the `ripple` value, for all other filters this sets the –3 dB point.

`order%` The order of the lowpass filter used as the basis of the design, in the range 1 to 10. If omitted, 2 is used. The order of the filter implemented is `order%*2`. High orders and narrow pass bands may lose numerical accuracy in the filter output.

`type%` Set 0 for Butterworth, 1 for Bessel and 2 for Chebyshev type 1, 3 for Chebyshev type 2. The default value is 0 for a Butterworth filter.

`ripple` The desired pass band ripple in dB for Chebyshev type 1 filters (default 3 dB) or the desired minimum cut in the stop band for Chebyshev type 2 filters (default 40 dB). The value here must be greater than 0.

`get%` The command variant with this argument returns information about the last filter you created with this command. See the discussion of IIR commands for details.

`arr` This is an option vector or matrix used to return information about the last filter you created with this command. See the discussion of IIR commands for details.

Returns All forms of the command return negative numbers for errors. The forms that apply or create filters return 0 for success. The other command forms have their return values included in the description of the `get%` argument or return 0.

See also: `FIRMake()`, `IIRBp()`, `IIRHp()`, `IIRLp()`, `IIRNotch()`, `IIRReson()`

IIRHp()

This function creates and applies IIR (Infinite Impulse Response) high pass filters to arrays of data. You can run the filter forwards or backwards through the data.

```
Func IIRHp(data[]|0, edge{, order%{, type%{, ripple}}});
Func IIRHp(data[]{, -1});
Func IIRHp(get%{, arr[]{[]}});
```

`data` An array of data to filter. If this is the only argument, or if there are 2 arguments and the second is negative, the last created high pass filter is used. Otherwise, the filter defined by the remaining arguments is used. Replace `data` with 0 to create a filter. Filters run forward through `data` unless there are 2 arguments and the second is negative, when the filter runs backwards.

edge The corner frequency of the high pass filter. This is expressed as a fraction of the sample rate and is limited to the range 0.0002 to 0.4998. For Chebyshev filters, this is the point at which the attenuation reaches the `ripple` value, for other filters this sets the –3 dB point.

`order%` The order of the filter in the range 1 to 10. If omitted, 2 is used.

`type%` Set 0 for Butterworth, 1 for Bessel and 2 for Chebyshev type 1, 3 for Chebyshev type 2. The default value is 0 for a Butterworth filter.

`ripple` The desired pass band ripple in dB for Chebyshev type 1 filters (default 3 dB) or the desired minimum cut in the stop bands for Chebyshev type 2 filters (default 40 dB). The value here must be greater than 0.

`get%` The command variant with this argument returns information about the last filter you created with this command. See the discussion of IIR commands for details.

`arr` This is an option vector or matrix used to return information about the last filter you created with this command. See the discussion of IIR commands for details.

Returns All forms of the command return negative numbers for errors. The forms that apply or create filters return 0 for success. The other command forms have their return values included in the description of the `get%` argument or return 0.

See also: `FIRMake()`, `IIRBp()`, `IIRBs()`, `IIRLp()`, `IIRNotch()`, `IIRReson()`

IIRLp()

This function creates and applies IIR (Infinite Impulse Response) low pass filters to arrays of data. You can run the filter forwards or backwards through the data.

```
Func IIRLp(data[]|0, edge{, order%{, type%{, ripple}}});
Func IIRLp(data[]{, -1});
Func IIRLp(get%{, arr[]{[]}});
```

`data` An array of data to filter. If this is the only argument, or if there are 2 arguments and the second is negative, the last created low pass filter is used. Otherwise, the filter defined by the remaining arguments is used. Replace `data` with 0 to create a filter. Filters run forward through `data` unless there are 2 arguments and the second is negative, when the filter runs backwards.

edge The corner frequency of the low pass filter. This is expressed as a fraction of the sample rate and is limited to the range 0.0002 to 0.4998. For Chebyshev filters, this is the point at which the attenuation reaches the `ripple` value, for other filters this sets the –3 dB point.

`order%` The order of the filter in the range 1 to 10. If omitted, 2 is used.

`type%` Set 0 for Butterworth, 1 for Bessel and 2 for Chebyshev type 1, 3 for Chebyshev type 2. The default value is 0 for a Butterworth filter.

`ripple` The desired pass band ripple in dB for Chebyshev type 1 filters (default 3 dB) or the desired minimum cut in the stop bands for Chebyshev type 2 filters (default 40 dB). The value here must be greater than 0.

`get%` The command variant with this argument returns information about the last filter you created with this command. See the discussion of IIR commands for details.

`arr` This is an option vector or matrix used to return information about the last filter you created with this command. See the discussion of IIR commands for details.

Returns All forms of the command return negative numbers for errors. The forms that apply or create filters return 0 for success. The other command forms have their return values included in the description of the `get%` argument or return 0.

See also:`FIRMake()`, `IIRBp()`, `IIRBs()`, `IIRHp()`, `IIRNotch()`, `IIRReson()`

IIRNotch()

This function creates and applies IIR (Infinite Impulse Response) notch filters to arrays of data. You can run the filter forwards or backwards through the data. The gain of the notch filter is zero at the notch frequency and 1 at low and high frequencies.

```
Func IIRNotch(data[]|0, fr, q);
Func IIRNotch(data[]{, -1});
Func IIRNotch(get%{, arr[]{[]}});
```

`data` An array of data to filter. If this is the only argument, or if there are 2 arguments and the second is negative, the last created low pass filter is used. Otherwise, the filter defined by the remaining arguments is used. Replace `data` with 0 to create a filter. Filters run forward through `data` unless there are 2 arguments and the second is negative, when the filter runs backwards.

`fr` The frequency of the notch. This is expressed as a fraction of the sample rate and is limited to the range 0.0002 to 0.4998.

`q` The desired `q` factor for the notch filter. If the two –3 dB points either side of the notch are at frequencies `Flo` and `Fhi`, q is given by `fr/(Fhi-Flo)`. The higher the q, the narrower the notch. Try 100 as a starting point.

`get%` The command variant with this argument returns information about the last filter you created with this command. See the discussion of IIR commands for details.

`arr` This is an option vector or matrix used to return information about the last filter you created with this command. See the discussion of IIR commands for details.

Returns All forms of the command return negative numbers for errors. The forms that apply or create filters return 0 for success. The other command forms have their return values included in the description of the `get%` argument or return 0.

See also: `FIRMake()`, `IIRBp()`, `IIRBs()`, `IIRHp()`, `IIRLp()`, `IIRReson()`

IIRReson()

This function creates and applies IIR (Infinite Impulse Response) resonator filters to arrays of data. You can run the filter forwards or backwards through the data. The gain of the filter is 1 at the resonator frequency and zero at low and high frequencies.

```
Func IIRReson(data[]|0, fr, q);
Func IIRReson(data[]{, -1});
Func IIRReson(get%{, arr[]{[]}});
```

`data` An array of data to filter. If this is the only argument, or if there are 2 arguments and the second is negative, the last created low pass filter is used. Otherwise, the filter defined by the remaining arguments is used. Replace `data` with 0 to create a filter. Filters run forward through `data` unless there are 2 arguments and the second is negative, when the filter runs backwards.

`fr` The centre frequency of the resonator. This is expressed as a fraction of the sample rate and is limited to the range 0.0002 to 0.4998.

`q` The desired `q` factor for the resonator. If the two –3 dB points either side of the resonance are at frequencies `Flo` and `Fhi`, q is given by `fr/(Fhi-Flo)`. The higher the q, the narrower the resonance. Try 100 as a starting point.

`get%` The command variant with this argument returns information about the last filter you created with this command. See the discussion of IIR commands for details.

`arr` This is an option vector or matrix used to return information about the last filter you created with this command. See the discussion of IIR commands for details.

Returns All forms of the command return negative numbers for errors. The forms that apply or create filters return 0 for success. The other command forms have their return values included in the description of the `get%` argument or return 0.

See also: `FIRMake()`, `IIRBp()`, `IIRBs()`, `IIRHp()`, `IIRLp()`, `IIRNotch()`

Input()

This function reads a number from the user. It opens a window with a message, and displays the initial value of a variable. You can limit the range of the result.

```
Func Input(text$, val {,low {,high}});
```

`text$` A string holding a prompt for the user. If the string contains a vertical bar character (|), the string before the bar will be used to set the title of the window.

`val` The initial value to be displayed for editing. If limits are given, and the initial value is outside the limits, it is set to the nearer limit.

`low` An optional low limit for the result. If `low>=high`, the limits are ignored.

`high` An optional high limit for the result.

Returns The value typed in. The function always returns a value. If an out-of-range value is entered, the function warns the user and a correct value must be given. When parsing the input, leading white space is ignored and the number interpretation stops at the first non-numeric character or the end of the string.

See also: `DlgReal()`, `DlgInteger()`, `Input$()`

Input$()

This function reads user input into a string variable. It opens a window with a message, and displays a string. You can also limit the range of acceptable characters.

```
Func Input$(text$, edit${, maxSz%{, legal$}});
```

`text$` A string holding a prompt for the user. If the string contains a vertical bar character (|), the string before the | sets the title of the window.

`edit$` The starting value for the text to edit.

`maxSz%` Optional, maximum size of the response string.

`legal$` A string holding the characters that are acceptable. The starting string is filtered before display. A hyphen indicates a range of characters. To include a hyphen in the list, place it first or last in the string. Upper and lower case characters are distinct. For alphanumeic characters use: `"a-zA-Z0-9"`.

If this string is omitted, all printing characters are allowed, equivalent to `" -~"` (space to tilde). For simple use, the sequence of printing characters is:

```
space !"#$%&'()*+,-./0123456789:;<=>?@
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`
abcdefghijklmnopqrstuvwxyz{|}~
```

If you use extended or accented characters, the order depends on the system.

Returns The result is the edited string. A blank string is a possible result.

See also: `DlgString() ,Input()`

InStr()

This function searches for a string within another string. This function is case sensitive.

```
Func InStr(text$, find$ {,index%});
```

`text$` The string to be searched.

`find$` The string to look for.

`index%` If present, the start character index for the search. The first character is index 1.

Returns The index of the first matched character, or 0 if the string is not found.

See also: `DelStr$(), LCase$(), Left$(), Mid$(), Right$(), UCase$()`

Interact()

This function provides a quick and easy way to interact with a user. Cursors can always be dragged as we assume that they are one of the main ways of interacting with the data. You can restrict the user to a single view and limit the menu commands that can be used.

```
Func Interact(msg$, allow% {,help {, lb1$ {,lb2$ {,lb3$...}}}});
```

`msg$` The prompt to display in the tool bar during the operation. If there is not enough space to display the message and buttons, the message is truncated.

`allow%` A code that specifies the actions that the user can and cannot take while interacting with Signal. The code is the sum of possible activities:

1	User may swap to other applications.
2	User may change the current window.
4	User may move and resize windows.
8	User may use the File menu.
16	User may use the Edit menu.
32	User may use the View menu.
64	User may use the Analysis menu.
128	User may use the Cursor menu and add cursors.
256	User may use the Window menu.
512	User may use the Sample menu.
1024	User may not double click y axis.
2048	User may not double click the x axis or scroll it.
4096	User may not change channel of horizontal cursors.
8192	User may not change to another frame.

A value of 0 would restrict the user to inspecting data and positioning cursors in a single, unmoveable window, but being able to switch frames. A value of 8192 is the same but without changing frames.

`help` This can be either a number or a string. If it is a number, it is the number of a help item (if help is supported). If it is a string, it is a help context string. This is used to set the help information that is presented when the user requests help in the manner supported on the host machine. Set 0 to accept the default help.

`lb1$` These label strings create buttons, from right to left, in the tool bar. If no labels are given, one label is displayed with the text `"OK"`. The maximum number of buttons is 17. Buttons can be linked to the keyboard using `&` and by adding a vertical bar followed by a key code to the end of the label. You can also set a tooltip. The format is `"Label|code|tip"`. See the documentation for `label$` in the `ToolbarSet()` command for details.

Returns The number of the button that was pressed. Buttons are numbered in order, so `lb1$` is button 1, `lb2$` is button 2 and so on.

With `allow%` set to 0, all the user could do would be to press a button on the tool bar. The tool bar would be displayed (if it was not present) when `Interact()` was called. When the user presses a button to exit, the tool bar is returned to the state it was in before Interact was used.

See also: `Toolbar()`

LastTime()

This function finds the first item on a channel before a particular x axis position.

```
Func LastTime(chan%, &pos{, &val|code%[]});
```

`chan%` The channel number (1 to n) to use for the search.

`pos` The x axis value to search before. Items at the position are ignored. To start a backward search that guarantees to iterate through all items, start at `Maxtime(chan%)+1`.

`pos` is updated to contain the x axis position of the previous item. It is left unchanged if no more items are found or there is an error.

`val` This optional parameter returns the waveform value for waveform channels.

`code%` This optional parameter is only used if the channel is a marker type. This is an array with at least four elements that is filled in with the marker codes.

Returns The function returns 1 if a data item is found, 0 if there are no more items to be found or a negative error code.

See also: `Maxtime(), Mintime(), NextTime()`

LCase$()

This function converts a string into lower case.

```
Func LCase$(text$);
```

`text$` The string to convert.

Returns A lower-cased version of the original string.

See also: `UCase$()`

Left$()

This function returns the first `n` characters of a string.

```
Func Left$(text$, n);
```

`text$` A string of text.

`n` The number of characters to extract.

Returns The first n characters, or all the string if it is less than `n` characters long.

See also: `DelStr$(), Mid$(), InStr(), Len(), Right$()`

Len()

This function returns the length of a string or the size of a one dimensional array.

```
Func Len(text$);
Func Len(arr[]);
```

`text$` The text string.

`arr[]` A one dimensional array. It is an error to pass in a two dimensional array.

Returns The length of the string or the array, as an integer.

You can find out the size of each dimension of a two dimensional array as follows:

```
proc something(arr[][])    'function passed a 2-d array
var n%, m%;
n% := Len(arr[][0]);       'get size of first dimension
m% := Len(arr[0][]);       'get size of second dimension
return;
end;
```

Ln()

This function calculates the natural logarithm (inverse of `Exp()`) of an expression, or replaces the elements of an array with their natural logarithms.

```
Func Ln(x|x[]{[]...});
```

`x` A real number or a real array. Zero or negative numbers cause the script to halt with an error unless the argument is an array, when an error code is returned.

Returns When used with an array, it returns 0 if all was well, or a negative error code. When used with an expression, it returns the natural logarithm of the argument.

LnGamma()

The natural logarithm of the gamma function $\Gamma(x)$ is available from the script language for real values of $x > 0.0$. The gamma function has the useful property that $\Gamma(n+1)$ is the same as $n!$ (n factorial) for integral values of n. However, the gamma function becomes inconveniently large, reaching floating-point infinity as far as the script language is concerned when x is 172.62. As this is a rather restricted range, the script returns the natural logarithm of the gamma function. The mathematical definition of the gamma function is:

$$\Gamma(a) = \int_0^\infty e^{-t} t^{a-1} dt$$

```
Func LnGamma(a);
```

`a` A positive value. The script stops with a fatal error if this is negative.

Returns The natural logarithm of the Gamma function of `a`.

Log()

Gives the logarithm to base 10 of the argument or replaces the elements of an array with their logarithms to base 10.

```
Func Log(x|x[]{[]...});
```

`x` A real number or a real array. Zero or negative numbers cause the script to halt with an error unless the argument is an array, when an error code is returned.

Returns With an array, this returns 0 if all was well or a negative error code. With an expression, this returns the logarithm of the number to the base 10.

LogHandle()

The log window, also called the log view, is a text view created by the application and is the destination for `PrintLog()`. This function returns the view handle of the log window. You need this if you are to size or hide the log window, or make it the current or front window, or use the editing commands to clear it.

```
Func LogHandle();
```

Returns The view handle of the log window.

See also: `EditClear()`, `EditSelectAll()`, `View()`, `FrontView()`, `Window()`, `WindowGetPos()`, `WindowSize()`, `WindowVisible()`

MarkCode()

This returns the data stored in a marker at a particular x axis position.

```
Func MarkCode(chan%, pos{, co%|co%[]});
```

`chan%` The marker channel to read.

`pos` The position of the marker. This must match to within ± half the time interval returned by `BinSize()` for the channel.

`co%` Optional integer to return the first 8-bit marker code (0 to 255).

`co%[]` Optional array in which to return the marker codes. Up to 4 of these are returned depending on the size of the array.

Returns The first code if a marker was found, or -1 if no marker exists at `pos`.

See also: `BinSize()`, `MarkEdit()`, `MarkTime()`

MarkEdit()

This changes the data stored in a marker at a particular x axis position.

```
Func MarkEdit(chan%, pos, co%|co%[]);
```

`chan%` The marker channel to edit.

`pos` The position of the marker. This must match to within ± half the time interval returned by `BinSize()` for the channel.

`co%` A value from 0 to 255 to replace the first code for the marker.

`co%[]` Array of up to 4 values (0 to 255) to replace codes for the marker. If the array size is smaller than 4 the other codes are left untouched.

Returns 0 if a marker was edited, or -1 if no marker exists at `pos`.

See also: `BinSize()`, `MarkCode()`, `MarkTime()`

MATDet()

This calculates the determinant of a matrix (a two dimensional array).

```
Func MATDet(mat[][]);
```

`mat` A two dimensional array with the same number of rows and columns.

Returns The determinant of `mat` or 0.0 if the matrix is singular.

See also: `ArrAdd()`, `MATInv()`, `MATMul()`, `MATTrans()`

MATInv()

This inverts a matrix (a two dimensional array) and optionally returns the determinant.

```
Func MATInv(inv[][] {,src[][]{, &det}});
```

`inv` A two dimensional array to hold the result. If `src` is omitted, `inv` is replaced by its own inverse. The number of rows and columns of `inv` must be the same.

`src` If present, the matrix to invert. The numbers of rows and columns of this two dimensional array must be at least as large as `inv`.

`det` If present, returned holding the determinant of the inverted matrix.

Returns 0 if all was OK, -1 if the matrix was singular or very close to singular.

See also: `ArrAdd()`, `MATMul()`

MATMul()

This function multiplies matrices (two dimensional arrays) and stores the result in a third. In matrix terms, this evaluates **A** = **BC** where **A** is an `m` rows by `n` columns matrix, **B** is an `m` by `p` matrix and **C** is a `p` by `n` matrix.

```
Proc MATMul(a[][], b[][], c[][]);
```

`a` A `m` by `n` matrix of reals to hold the result.

`b` A `m` by `p` matrix.

`c` A `p` by `n` matrix.

See also: `ArrMul(), MATInv()`

MATSolve()

This function solves the matrix equation **Ax** = **y** for **x**, given **A** and **y**. Both **x** and **y** are vectors of length `n` and **A** is an `n` by `n` matrix.

```
Func MATSolve(x[], a[][], y[]);
```

`x` A one dimensional real array of length `n` to hold the result.

`a` A two dimensional (`n` by `n`) array of reals holding the matrix.

`y` A one dimensional real array of length `n`.

Returns The functions returns 0 if all is OK or -1 if `a` is a singular matrix.

See also: `ArrMul(), MATInv()`

MATTrans()

This transposes a matrix (a two dimensional array), swapping the rows and columns.

```
Proc MATTrans(mat[][]{, src[][]});
```

`mat` A two dimensional `n` by `m` array returned holding the transpose of `src`. If `src` is omitted, `m` must be equal to `n` and the rows and columns of `mat` are swapped.

`src` Optional, a two dimensional `m` by `n` array to transpose.

See also: `ArrAdd(), MATMul()`

MarkTime()

This reads and changes the time for a marker.

```
Func MarkTime(chan%, pos, {new});
```

`chan%` The channel number holding markers to move.

`pos` The position of the marker. This must match to within ± half the time interval returned by `BinSize()` for the marker channel.

`new` If supplied, the new position (x axis value) for the marker. Note that marker times must be in order, so this time will be truncated to prevent the marker time reaching or going past adjacent markers.

Returns The exact marker time before any changes or 0 if no marker exists at `pos`.

See also: `BinSize(), MarkCode(), MarkEdit(), View(v,c).[]`

Max()

This function returns the maximum of several real and/or integer variables or the index of the maximum value in an array if an array argument is provided. See `Min()` for example.

```
Func Max(arr[]|arr%[]|val1 {,val2 {,val3...}});
```

`arr` A real or integer array.

`valn` A list of real and/or integer values to scan for a maximum.

Returns The maximum value or index of the maximum in an array argument.

See also: `Min(), MinMax(), XYRange`

Maxtime()

This returns the maximum x axis value in the frame or a specified channel, or the latest time reached within the frame or the specified channel in a sampling document view. For the end of the visible x axis use `XHigh()`.

```
Func MaxTime({chan%});
```

`chan%` An optional channel number (1 to n). If present, and if the channel exists, the function gets the x axis value for the last item sampled in the channel or the maximum x axis value in the frame if no items are found on the channel, or if no channel was specified. If `chan%` is zero, the value returned is the frame length limit – the maximum X axis value for a frame regardless of the points that happen to be currently stored – this is useful for frame 0 of a file being sampled.

Returns The value returned is the maximum x axis value for the frame or a specified channel. If the current view is of the wrong type, or if a specified channel number does not exist, the script stops with an error.

See also: `Len(), LastTime(), NextTime(), Mintime(), Seconds(), XHigh()`

Message()

This function displays a message in a box with an OK button that the user must click to remove the message. Alternatively, the user can press the Enter key.

```
Proc Message(form$ {,arg1 {,arg2...}});
```

`form$` A string that defines the output format as for `Print()`. If the string contains a vertical bar character (|) then that portion of the string before the | will be used to set the title of the dialog box.

`arg1,2` The arguments used to replace `%d %f` and `%s` type formats.

The output string will be presented as one line if it is short enough, otherwise it will be split into multiple lines. Messages longer than 70 characters are truncated.

See also: `Print(), Input(), Query(), DlgCreate()`

Mid$()

This function returns a substring of a string.

```
Func Mid$(text$, index% {,count%});
```

`text$` A text string.

`index%` The starting character in the string. The first character is index 1.

`count%` The maximum characters to return. If omitted, all the characters to the end of the string are returned.

Returns The specified string. If `index%` is larger than the original string length, the result is an empty string.

See also: `DelStr$(), InStr(),Left$(), Len(), Right$(),`

Min()

This function returns the minimum of several real and/or integer variables or the index of the minimum value in an array if an array argument is provided.

```
Func Min(arr[]|arr%[]|val1 {,val2 {,val3...}});
```

`arr` A real or integer array.

`valn` A list of real and/or integer values to scan for a minimum.

Returns The minimum value or index of the minimum in an array argument.

An example finding the minimum in a sub-array holding 10 items of the original data:

```
var data[70], minPos%, minVal;
...
minPos:=Min(data[40:10]);      ' returns a position between 0 and 9
minVal:=data[40+minPos];                          ' value of minimum
```

See also: `Max(), Minmax(), XYRange`

Minmax()

`Minmax()` finds the minimum and maximum values for data view channels with a y axis, or the minimum and maximum intervals for a marker channel handled as dots or lines. The values returned for marker channels as a rate histogram are measured from the histogram with partial bins included.

```
Func Minmax(chan%, start, finish, &min, &max {,&minP{,&maxP
                                            {,mode% {,binsz}}}});
```

`chan%` The channel number (1 to n) on which to find the maximum and minimum.

`start` The start position in x axis units.

`finish` The end position in x axis units.

`min` The minimum value is returned in this variable or zero if no data found.

`max` The maximum value is returned in this variable or zero if no data found.

`minP` The position of the minimum is returned in this variable or zero if no data found.

`maxP` The position of the maximum is returned in this variable or zero if no data found.

`mode%` This will have no effect for a waveform channel. If present for a marker channel, this sets the effective drawing mode in which to find the minimum and maximum. If `mode%` is absent or inappropriate, the current display mode is used. The modes are:

0 The current mode for the channel. Any additional arguments are ignored.
1 Dots mode for markers, returns the position of the marker at or after `pos`.
2 Lines mode for markers, result is the same as mode 1.
3 Rate mode for markers. The `binSz` argument sets the width of each bin.

`binSz` This sets the width of the rate histogram bins when specifying rate mode.

Returns 1 if data points were found, 0 if no data was found or a negative error code.

See also: `Min(), Max(), View(v,c).[], XYRange()`

Mintime()

In a data view, this returns the minimum x axis value in the frame or in a channel. For the end of the visible x axis use `XLow()`.

`Func MinTime({chan%});`

`chan%` An optional channel number (1 to n). If present, and if the channel exists, the function gets the x axis value for the earliest item in the channel or the minimum x axis value in the frame if no items are found on the channel or if no channel was specified. If `chan%` is zero, the value returned is the frame length limit – the minimum X axis value for a frame regardless of the points that happen to be currently stored – this is useful for frame 0 of a file being sampled.

Returns The value returned is the minimum x axis value in the frame or channel. If the current view is of the wrong type, or if the channel number is illegal the script stops with an error.

See also: `Len(), BinZero(), ChanRange(), LastTime(), NextTime(), Maxtime(), Seconds(), XLow()`

MoveBy()

This moves the text caret in a text view relative to the current position. You move the caret by lines and/or a character offset. You can extend or cancel the current selection.

`Func MoveBy(sel%, char%{, line%});`

`sel%` If zero, all selections are cleared. If non-zero the selection is extended to the destination of the move and the new position is the start of the selection.

`char%` This is a character offset. If `line%` is absent, the new position is obtained by adding `char%` to the current position. If this is beyond the start or end of the text it is limited to the start or end.

`line%` If present it specifies a line offset to apply. To find the new position add `line%` to the current line number and `char%` to the current character position in the line. If the new line number is beyond the start or end of the text it is limited to the start and end. If the new character position is beyond the start or end of the line it is limited to the start or end of the line.

Returns The function returns the new position in the file of the start of the selection. `MoveBy(1,0)` finds the current position without changing the selection.

See also: `MoveTo(), Selection$()`

MoveTo()

This moves the text caret in a text view. You position the caret by lines and/or a character offset. You can extend or cancel the current selection.

`Func MoveTo(sel%, char%{, line%});`

`sel%` If zero, all selections are cleared. If non-zero the selection is extended to the destination of the move and the new current position is the start of the selection.

`char%` This is a character offset. If `line%` is absent, this sets the new position in the file. If this is beyond the start or end of the text it is limited to the start or end. A position of 0 places the caret before the first character of the first line.

`line%` If present it specifies the new line number. If it is beyond the start or end of the text it is limited to the start and end. If the new character position is beyond the start or end of the line it is limited to the start or end of the line.

Returns The function returns the new position in the file of the start of the selection.

See also: `MoveBy(), Selection$()`

NextTime()

This function is used to find the next item on a channel after a particular x axis position.

```
Func NextTime(chan%, &pos{,&val|code%[]});
```

`chan%` The channel number (1 to n) to use for the search.

`pos` The x axis position to start the search after. Items at the position are ignored. To ensure that items at the `Mintime()` are found, set position to `Mintime()-1`. `pos` is updated to contain the x axis position of the next item. It is left unchanged if no more items are found or there is an error.

`val` This optional argument is used with waveform channels. It is returned holding the waveform value.

`code%` This optional parameter is only used if the channel is a marker type. This is an array with at least four elements that is filled in with the marker codes.

Returns The function returns 1 if a data item is found, 0 if there are no more items to be found, or a negative error code.

See also: `LastTime(), MaxTime(), MinTime()`

OpClEventGet()

This function gets the details of a particular event in an idealised trace.

```
Func OpClEventGet(chan%, meth%, &time{, &period{, &amp{,
                  &flags%}}});
```

`chan%` The channel number of the idealised trace.

`meth%` The indexing method used to determine which event we are addessing. Possible values are:

- 0 Use the selected event and ignore the `time` parameter.
- 1 Find the first event starting before the specified time or the first event in the trace if none exist before `time`.
- 2 Find the event with a start time closest to the specified time.
- 3 Find the first event starting after the specified time.

`time` The time used to address the event. This will be set to the start time of the event if an event is found.

`period` This will be set to the duration of the event if found.

`amp` This will be set to the amplitude of the event if found.

`flags%` This will contain the events flags if found. A full list of flags can be found in the description of `SetOpClHist()`.

Returns The function returns 1 if an event was found otherwise it returns 0.

See also: `OpClEventSet(), OpClEventDelete(), OpClEventSplit(), SetOpClHist()`

OpClEventChop()

This function splits the specified event into two, each having a period equal to half that of the original. If the event has an amplitude between those of the preceding and following events then the amplitudes and flags of the first and second new events will be taken from the following and preceding events respectively. In this case the new events will also be flagged as having assumed amplitudes.

```
Func OpClEventChop(chan%, meth%{, time});
```

`chan%` The channel number of the idealised trace.

`meth%` The indexing method used to determine which event we are addessing. This is the same as for `OpClEventGet()`.

`time` The time used to address the event (not needed if `meth%` is 0).

Returns The function returns 1 if an event is found otherwise it returns 0.

See also: `OpClEventDelete(), OpClEventGet(), OpClEventMerge(), OpClEventSet(), OpClEventSplit()`

OpClEventDelete()

This function deletes a specified event from an idealised trace and amalgamates its neighbours to produce a single event covering the time range of all three events and an amplitude caluculated as an average of the amplitudes of all three events weighted by their durations. The flags for this event will be taken from the earliest of the three original events.

```
Func OpClEventDelete(chan%, meth%{, time});
```

`chan%` The channel number of the idealised trace.

`meth%` The indexing method used to determine which event we are addessing. This is the same as for `OpClEventGet()`.

`time` The time used to address the event (not needed if `meth%` is 0).

Returns The function returns 1 if an event was found and deleted, otherwise it returns 0.

See also: `OpClEventGet(),OpClEventSet(),OpClEventSplit()`

OpClEventMerge()

This function amalgamates the specified event with the event to the right to produce a single event covering the time range of both events and an amplitude calculated as an average of the amplitudes of both events weighted by their durations. The flags for this event will be taken from the original specified event.

```
Func OpClEventMerge(chan%, meth%{, time});
```

`chan%` The channel number of the idealised trace.

`meth%` The indexing method used to determine which event we are addessing. This is the same as for `OpClEventGet()`.

`time` The time used to address the event (not needed if `meth%` is 0).

Returns The function returns 1 if an event was found and merged, otherwise it returns 0.

See also: `OpClEventChop(), OpClEventDelete(), OpClEventGet(), OpClEventSet(), OpClEventSplit()`

OpClEventSet()

This function sets the details of a particular event in an idealised trace. Neighbouring events will be adjusted to accommodate the new values by altering the start time or duration acordingly. If you attempt to modify an event beyond the time range of the immediate neighbours then the function will fail and 0 will be returned.

```
Func OpClEventSet(chan%, meth%, time, start, period{, amp{,
                    flags%}});
```

`chan%` The channel number of the idealised trace.

`meth%` The indexing method used to determine which event we are addessing. This is the same as for `OpClEventGet()`.

`time` The time used to address the event.

`start` The new start time for the event being indexed.

`period` The new duration for the event.

`amp` The new amplitude for the event. If omited the amplitude will be left unchanged.

`flags%` The new flag values. If omitted the flags are left unchanged. A full list of flags can be found in the description of `SetOpClHist()`.

Returns The function returns 1 if an event was found and set, otherwise it returns 0.

See also: `OpClEventGet()`,`OpClEventDelete()`,`OpClEventSplit()`, `SetOpClHist()`

OpClEventSplit()

This function splits the specified event into three, each having a period equal to one third that of the original.

```
Func OpClEventSplit(chan%, meth%{, time});
```

`chan%` The channel number of the idealised trace.

`meth%` The indexing method used to determine which event we are addessing. This is the same as for `OpClEventGet()`.

`time` The time used to address the event (not needed if `meth%` is 0).

Returns The function returns 1 if a data item is found, 0 if there are no more items to be found, or a negative error code.

See also: `OpClEventGet()`,`OpClEventSet()`,`OpClEventDelete()`

OpClFitRange()

This function fits an idealised trace so that the convolution with the step response function of the filter used to sample the original data overlays the observed raw data trace.

```
Func OpClFitRange(chan%, start, end);
```

`chan%` The channel number of the idealised trace.

`start` The start time of the range to fit.

`end` The end time of the range to fit

Returns The function returns 1 if a data item is found, 0 if there are no more items to be found, or a negative error code.

See also: `OpClEventChop()`, `OpClEventDelete()`, `OpClEventGet()`, `OpClEventMerge()`, `OpClEventSet()`

Optimise()

This has the same effect as the optimise button in the YAxis dialog and can be used in a data or XY view. Optimising a channel that is not displayed is not an error. If you give a channel number that is not displayed, we assume that you know what you are doing, so it is optimised in the display mode that would be used if the channel were turned on.

```
Proc Optimise(cSpc{, start{, finish}});
```

`cSpc` A channel list specifier of the channels to optimise. See the *Script language syntax* chapter for a definition of channel specifiers.

`start` The start of the region to optimise. If omitted, this is the start of the window.

`finish` The end of the region to optimise. If omitted, this is the end of the window.

See also: `YRange(), YLow(), YHigh(), MinMax(), XYRange()`

Overdraw()

This function turns overdraw mode on and off for the current view. It also returns the current overdraw mode. With overdraw mode on, a view will display not only the current frame, but also all of the other frames in the overdraw frame list.

```
Func Overdraw({on%{, cycle%}});
```

`on%` Optional argument that sets overdraw mode. If this is omitted, no change is made.

0 Turn overdraw off.
1 Turn overdraw on.

`cycle%` Optional argument that turns on colour cycling if present and non-zero. If omitted or zero, colour cycling is turned off.

Returns The state of overdraw mode at the time of the call, or a negative error code. Changes made by this function do not cause an immediate redraw.

See also: `OverdrawFrames(), OverdrawGetFrames()`

OverdrawFrames()

This function is used to set or modify the list of frames to overdraw in the data view. You can specify a range of frame numbers or a list of frames. If the function is used with no arguments it clears the overdraw frame list.

```
Func OverdrawFrames({sFrm%{, eFrm%{, mode%{, add%}}}});
Func OverdrawFrames(frm$|frm%[]{, mode%{, add%}});
```

`sFrm%` First frame to include. This option processes a range of frames. `sFrm%` can also be a negative code as follows:

-101 The overdraw list is cleared and the frame buffer is added.
0 All sampled frames (on-line only).
-1 All frames in the file are included.
-2 The frame current at the time of this call.
-3 Frames must be tagged.
-5 Last N frames (on-line only).
-6 Frames must be untagged.

Choosing a negative code with `add%` set to 0 will allow Signal to modify the overdraw status of individual frames as they are subsequently tagged/untagged. etc If this command is used with add% absent or set to non-zero then this dynamic behaviour will be lost.

`eFrm%` Last frame to include. If this is -1 the last frame is the last in the data view. This argument is ignored if `sFrm%` is less than 1.

`frm$` A frame specification string. This option specifies a list of frames using a string such as “1..32,40,50”.

`frm%[]` An array of frame numbers. This option provides a list of frame numbers in an array, the first array element holding the number of frames in the list.

`mode%` If `mode%` is present it is used to supply an additional criterion for including each frame in the range, list or specification. If `mode%` is absent all frames are included. As with `sFrm%`, these modes will be applied dynamicaly if `add%` is 0. If `sFrm%` is –5 then this is the value of N. If `sFrm%` is 0 then this value is ignored, otherwise the modes are:

0-n Frames must have a state matching the value of `mode%`.
-1 All frames in the range, list are included.
-2 Only the frame current at the time of drawing, if in the list, is included.
-3 Frames must be tagged.
-6 Frames must be untagged.

`add%` If `add%` is present and non-zero it determines whether the specified frames are to be added to or removed from the existing display list for the view, as follows:

-1 Remove the frames from the existing display list.
0 Clear the display list before adding these frames.
1 Add the frames to the existing display list (the default).

If `add%` is absent the new frame list will be added to the existing display list.

Returns The number of frames in the new overdraw list or a negative error code.

See also: `Overdraw()`, `OverdrawGetFrames()`

OverdrawGetFrames()

This function is used to get the list of frames overdrawn in the data view.

```
Func OverdrawGetFrames({list%[]});
```

`list%` An optional array of frame numbers to hold the list of frame numbers. If the array is too short, enough frames are returned to fill the array. Element zero holds the number of frames returned in the array.

Returns The number of frames that would be returned if the array was of unlimited length, or zero if the view is not a data view.

See also: `Overdraw(), OverdrawFrames()`

PaletteGet()

This reads back the percentages of red, green and blue in a colour in the palette.

```
Proc PaletteGet(col%, &red, &green, &blue);
```

`col%` The colour index in the palette in the range 0 to 39. Items 0 and 1 are permanently fixed as white and black.

`red` The percentage of red in the colour.

`green` The percentage of green in the colour.

`blue` The percentage of blue in the colour.

See also: `Colour(), PaletteSet()`

PaletteSet()

This call sets the colour of one of the palette items. There are 40 palette colours, numbered 0 to 39. Colours 0 to 6 in the palette are fixed, providing grey scales from black to white, and the rest can be specified. This command is the equivalent of mixing the colour by hand from the colour menu.

Colours are specified using the RGB (Red Green Blue) colour model. For example, bright blue is achieved by 0% red, 0% green and 100% blue. Bright yellow is 100% red, 100% green and 0% blue. Black is 0% of all three colours, white is 100% of all three colours. All screen pixels of a “solid” colour are the same hue. Systems with limited colour capabilities generate non-solid colours by mixing pixels of different hues.

```
Proc PaletteSet(col%, red, green, blue {,solid%});
```

`col%` The colour index in the palette in the range 0 to 39. Attempting to change a fixed colour, or a non-existent colour, has no effect.

`red` The percentage of red in the colour.

`green` The percentage of green in the colour.

`blue` The percentage of blue in the colour.

`solid%` If present and non-zero, the system sets the nearest solid colour to the colour requested. Systems that cannot or don't need to do this ignore the argument.

See also: `Colour(), PaletteGet()`

PCA()

This command performs Principal Component Analysis on a matrix of data. This can take a long time if the input matrix is large.

```
Func PCA(flags%, x[][], w[]{, v[][]});
```

`flags%` Add the following values to control pre-processing of the input data:

1 Subtract the mean value of each row from each row.
2 Normalise each row to have mean 0.0 and variance 1.0.
4 Subtract the mean value of each column from each column.
8 Normalise each column to have mean 0.0 and variance 1.0.

You would normally pre-process the rows or the columns, not both. If you set flags for both, the rows are processed first.

`x[][]` An `m` rows by `n` columns matrix of input data that is replaced by the output data. The first array index is the rows; the second is the columns. There must be at least as many rows as columns (`m` >= `n`). If you have insufficient data you can use a square matrix and fill the missing rows with zeros. If you were computing the principal components of spike data, on input each row would be a spike waveform. On output, each row holds the proportion of each of the `n` principal components scaled by the `w[]` array that, when added together, would best (in a least-squares error sense) represent the input data.

`w[]` This is an array of length at least `n` that is returned holding the variance of the input data that each component accounts for. The components are ordered such that `w[i] >= w[i+1]`.

`v[][]` This is an optional square matrix of size `n` by `n` that is returned holding the `n` principal components in the rows.

Returns 0 if the function succeeded, -1 if `m` < `n`, -2 if `w` has less than `n` elements or `v` has less than `n` rows or columns.

Pow()

This is the power function that raises `x` to the power of `y`. If an array is used then each element of the array is replaced with its value to the power of y.

```
Func Pow(x|x[]{[]...}, y);
```

`x` A real number or a real array to be raised to the power of `y`.

`y` The exponent. If `x` is negative, `y` must be integral.

Returns If x is an array, it returns 0 or a negative error code. If x is a number, it returns `x` to the power of `y` unless an error is detected, when the script halts.

Print()

This command prints to the current view, which must be a text view. The output is inserted at the position of the caret. For the commands equivalent to File menu Print options refer to `FilePrint()`.

If the first argument is a string (not an array), it is assumed to hold format information for the remaining arguments. If the first argument is an array or not a string or if there are more arguments than format specifiers, Signal prints the arguments without a format specifier in a standard format and adds a new line character at the end. If you provide a format string and you require a new line character at the end of the output, include `\n` at the end of the format string.

```
Func Print(form$|arg0 {,arg1 {,arg2...}});
```

`form$` A string that specifies how to treat the arguments that follow. The string contains two types of characters: ordinary text that is copied to the output unchanged and format specifiers that determine how to convert each of the following arguments to text. The format specifiers are introduced by a `%` and terminated by one of the letters `d`, `x`, `c`, `s`, `f`, `e` or `g` in upper or lower case. To place a literal `%` in the output, place `%%` in the format string.

`arg1,2` The arguments used to replace `%c`, `%d`, `%e`, `%f`, `%g`, `%s` and `%x` type formats.

Returns 0 or a negative error code. Fields that cannot be printed are filled with asterisks.

Format specifiers The full format specifier is:

`%{flags}{width}{.precision}format`

`flags` The `flags` are optional and can be placed in any order. They are single characters that modify the format specification as follows:

- `-` Specifies that the converted argument is left justified in the output field.
- `+` Valid for numbers, and specifies that positive numbers have a + sign.
- *`space`* If the first character of a field is not a sign, a space is added.
- `0` For numbers, causes the output to be padded to the field width on the left with 0.
- `#` For `x` format, `0x` is prefixed to non-zero arguments. For `e`, `f` and `g` formats, the output always has a decimal point. For `g` formats, trailing zeros are not removed.

`width` If this is omitted, the output field will be as wide as is required to express the argument. If this is present, it is a number that sets the minimum width of the output field. If the output is narrower than this, the field is padded on the left (on the right if the `-` flag was used) to this width with spaces (zeros if the `0` flag was used). The maximum width for numbers is 100.

`precision` This number sets the maximum number of characters to be printed for a string, the number of digits after the decimal point for `e` and `f` formats, the number of significant figures for `g` format and the minimum number of digits for `d` format (leading zeros are added if required). It is ignored for `c` format. There is no limit to the size of a string. Numeric fields have a maximum `precision` value of 100.

`format` The format character determines how the argument is converted into text. Both upper or lower-cased version of the format character can be given. If the formatting contains alphabetic characters (for example the `e` in an exponent, or hexadecimal digits `a-f`), if the formatting character is given in upper case the output becomes upper case too (`e+23` and `0x23ab` become `E+23` and `0X23AB`). The formats are:

`c` The argument is printed as a single character. If the argument is a numeric type, it is converted to an integer, then the low byte of the integer (this is equivalent to `integer mod 256`) is converted to the equivalent ASCII character. You can use this to insert control codes into the output. If the argument is a string, the first character of the string is output. The following example prints two tab characters, the first using the standard tab escape, the second with the ASCII code for tab (8):

```
Print("\t%c", 8);
```

`d` The argument must be a numeric type and is printed as a decimal integer with no decimal point. If a string is passed as an argument the field is filled with asterisks. The following prints “`  23,0002`”:

```
Print("%4d,%.4d", 23, 2.3);
```

`e` The argument must be a numeric type, otherwise the field is filled with asterisks. The argument is printed as `{-}m.dddddde±xx{x}` where the number of `d`'s is set by the precision (which defaults to 6). A precision of 0 suppresses the decimal point unless the `#` flag is used. The exponent has at least 2 digits (in some implementations of Signal there may always be 3 digits, others use 2 digits unless 3 are required). The following prints "`2.300000e+01,2.3E+00`":

```
Print("%4e,%.1E", 23, 2.3);
```

`f` The argument must be a numeric type, otherwise the field is filled with asterisks. The argument is printed as `{-}mmm.ddd` with the number of `d`'s set by the precision (which defaults to 6) and the number of `m`'s set by the size of the number. A precision of 0 suppresses the decimal point unless the `#` flag is used. The following prints "`+23.000000,0002.3`":

```
Print("%+f,%06.1f", 23, 2.3);
```

`g` The argument must be a numeric type, otherwise the field is filled with asterisks. This uses `e` format if the exponent is less than -4 or greater than or equal to the precision, otherwise `f` format is used. Trailing zeros and a trailing decimal point are not printed unless the `#` flag is used. The following prints "`2.3e-06,2.300000`":

```
Print("%g,%#g", 0.0000023, 2.3);
```

`s` The argument must be a string, otherwise the field is filled with asterisks.

`x` The argument must be a numeric type and is printed as a hexadecimal integer with no leading `0x` unless the `#` flag is used. The following prints "`1f,0X001F`":

```
Print("%x,%#.4X", 31, 31);
```

Arrays in the argument list

The `d`, `e`, `f`, `g`, `s` and `x` formats support arrays. One dimensional arrays have elements separated by commas; two dimensional arrays use commas for columns and new lines for rows. Extra new lines separate higher dimensions. If there is a format string, the matching format specifier is applied to all elements.

See also: `FilePrint(), Message(), ToolbarText(), Print$(), PrintLog()`

Print$()

This command prints formatted output into a string. The syntax is identical to the `Print()` command, but the function returns the generated output as a string.

```
Func Print$(form$|arg0 {,arg1 {,arg2...}});
```

`form$` An optional string that specifies how to format the arguments that follow. See `Print()` for a full description.

`arg1,2` The data to form into a string.

Returns It returns the string that is the result of the formatting operation. Fields that cannot be printed are filled with asterisks.

See also: `FilePrint(), Print(), PrintLog(), ReadStr()`

PrintLog()

This command prints to the log window. The syntax is identical to the `Print()` command, except that the output always goes to the log window and is always placed at the end of the view contents.

```
Func PrintLog(form$|arg0 {,arg1 {,arg2...}});
```

`form$` An optional string that specifies how to format the arguments that follow. See `Print()` for a full description.

`arg1,2` The data to print.

Returns 0 or a negative error code. Fields that cannot be printed filled with asterisks.

See also: `Print(), Print$(), Message()`

Process()

This function processes data into the current memory or XY view or to an idealised trace channel in the current data view. The view or channel must have been derived using `SetXXXX()` from a source data view which must not have been closed. This function takes data starting from a specified position in the current frame in the source data view and processes it.

```
Func Process(start {,clear% {,opt% {,optx% {,chan%}}}});
```

`start` The source view x axis position from which to start processing. Positions less than `MinTime()` are treated as `MinTime()`. Positions greater than `Maxtime()` mean no processing is done. The offset from the start of frame in `SetXXX()` will be ignored. If the start position specified plus the width of data required goes past the end of the source data then no data is processed.

`clear%` If present, and non-zero, the memory view bins are cleared before the results of the analysis are added to the view and `Sweeps()` result is reset.

`opt%` If present, and non-zero, the display of data in the memory view is optimised after processing the data.

`optx%` For XY views only, if present and non-zero, the X axis in the XY view is optimised after processing the data.

`chan%` For idealised trace channels only, if present, it is the channel number to process to.

Returns One if all is OK, zero if no data was processed or a negative error code.

A common mistake is to forget that the current view is not the source view and to use View`(0).xxx` when `View(ViewSource()).xxx` was intended.

See also: `SetXXX(), SetAverage(), SetPower(),ProcessAll(), ProcessFrames(), ProcessOnline(), Sweeps(), Optimise()`

ProcessAll()

This function is used in a data view to process all memory views or idealised trace channels derived from it.

```
Func ProcessAll(sFrm%{, eFrm%{, chans%}});
```

`sFrm%` The first frame to process.

`eFrm%` If this is present, a range of frames is processed, from `sFrm%` to `eFrm%` inclusive. If omitted only `sFrm%` is processed.

`chans%` If this is present and set to –1 then all idealised trace channels will be processed otherwise all memory views will be processed.

For each derived memory view, the settings of the `clear%` and `opt%` arguments are taken from the last call of `Process()` or `ProcessFrames()`. If a memory view had not yet been processed `clear%` is zero and `opt%` is non-zero.

Returns Zero if no errors or a negative error code.

See also: `Process(), ProcessFrames(), ProcessOnline()`

ProcessFrames()

This function is used in a derived memory view to process more than one frame from the source view. You can process a range of frame numbers or specify a list of frames.

```
Func ProcessFrames(sF% {,eF% {,mode%{,clear% {,opt% {,optx%
                                            {,chan%}}}}}});
Func ProcessFrames(frm$|frm%[]{,mode%{,clear% {,opt%
                                            {,optx%{,chan%}}}}});
```

`sF%` First frame to process. This option processes a range of frames. `sFrm%` can also be a negative code as follows:

- -1 All frames in the file are included.
- -2 The current frame.
- -3 Frames must be tagged.
- -6 Frames must be untagged.

`eF%` Last frame to process. If this is -1 the last frame in the data view is used. This argument is ignored if `sF%` is a negative code.

`frm$` A frame specification string. This option specifies a list of frames using a string such as "1..32,40,50".

`frm%[]` An array of frame numbers to process. This option provides a list of frame numbers in an array. The first element in the array holds the number of frames in the list.

`mode%` If `mode%` is present, it is used to supply an additional criterion for including each frame in the range, list or specification. If `mode%` is absent all frames are included. The modes are:

- 0-n Frames must have a state matching the value of `mode%`.
- -1 All frames in the specification are processed.
- -2 Only the current frame, if in the list, will be processed.
- -3 Frames must also be tagged.
- -6 Frames must also be untagged.

`clear%` If present and non-zero, the memory view bins are cleared before the results of processing the frames are added to the view and `Sweeps()` result is reset.

`opt%` If present and non-zero, the display of data in the memory view is optimised after processing the data.

`optx%` For XY views only, if present and non-zero, the X axis in the XY view is optimised after processing the data.

`chan%` For idealised trace channels only. This is the channel number to process to.

Returns Zero if no errors or a negative error code.

See also: `Process(), ProcessAll(), ProcessOnline()`

ProcessOnline()

This function is equivalent to the process dialog for a memory view derived from a sampling document view. It does not cause any processing, but sets up processing so that when the memory view for which the function is used is given a chance to update, the parameters set by this command are used.

```
Func ProcessOnline(mode% {,val% {,up% {,opt% {,optx% {,chan%
                                              (,clear%}}}}}});
```

`mode%` The modes are:

- 0 All sampled sweeps are processed regardless of whether they are written to disk. This mode will not work if you are using Fast triggers or Fast Fixed int sampling modes.
- -1 All sweeps saved to disk are processed.
- -3 All tagged frames written to disk are processed.
- -4 Sweeps with a state of `val%` are processed. A state of 0 is used if `val%` is not provided.
- -5 Processes the last `val%` sweeps including the latest. The result is cleared and `Sweeps()` count is reset to 0 before each update.
- -6 All untagged frames written to disk are processed.

`val%` In `mode%` -4 or -5 this provides the value for state or number of frames, respectively.

`up%` This provides the number of frames before the next process or zero for no gap.

`opt%` If present and non-zero, the memory view display is optimised after each process.

`optx%` For XY views only, if present and non-zero, the X axis in the XY view is optimised after processing the data.

`chan%` For idealised trace channels only. This is the channel number to process to.

`clear%` Set to 1 to clear bins before each process. Set to 0 or omit to leave bins unchanged before each process.

Returns 0 or a negative error code.

See also: `Process(), ProcessAll(), ProcessFrames()`

Profile()

Signal stores information within the `HKEY_CURRENT_USER\Software\CED\Signal` section of the system registry. The registry is organised as a tree of keys with lists of values attached to each key. If you think of the registry as a filing system, the keys are folders and the values are files. Keys and values are identified by case-insensitive text strings. This command can create and delete keys and store and read integer and string values, but only within the Signal section of the registry.

You can view and edit the registry with the `regedt32` program, which is part of your system. Select Run from the start menu and type `regedt32`, then click OK. Please read the `regedt32` help information before making any registry changes. It is a very powerful program; careless use can severely damage your system.

Do not write vast quantities of data into the registry; it is a system resource and should be treated with respect. If you must save a lot of data, write it to a text or binary file and save the file name in the registry. If you think that you may have messed up the Signal section of the registry, use `regedt32` to locate the `Signal` section and delete it. The next time you run Signal the section will be restored; you will lose any preferences you had set.

```
Proc Profile(key${, name${, val%{, &read%}}});
Proc Profile(key${, name${, val${, &read$}}});
```

`key$` This string sets the key to work on inside the `Signal` section of the registry. If you use an empty string, the `Signal` key is used. You can use nested keys separated by a backslash, for example `"My bit\\stuff"` to use the key `stuff` inside the key `My bit`. The key name may not start with a backslash. Remember that you must use two backslashes inside quote marks; a single backslash is an escape character. It is never an error to refer to a key that does not exist; the system creates missing keys for you.

`name$` This string identifies the data in the key to read or write. If you set an empty name, this refers to the `(default)` data item for the key set by `key$`.

`val` This can be either a string or an integer value. If `read` is omitted, this is the value to write to the registry. If `read` is present, this is the default value to return if the registry item does not exist.

`read` If present, it must have the same type as `val`. This is a variable that is set to the value held in the registry. If the value is not found in the registry, the variable is set to the value of the `val` argument.

`Profile()` can be used with 1 to 4 arguments. It has a different function in each case:

1 The key identified by `key$` is deleted. All sub-keys and data values attached to the key and sub-keys are also deleted. Nothing is done if `key$` is empty.
2 The value identified by `name$` in the key `key$` is deleted.
3 The value identified by `name$` in the key `key$` is set to `val%` or `val$`.
4 The value identified by `name$` in the key `key$` is returned in `val%` or `val$`.

The following script example collects values at the start, then saves them at the end:

```
var path$, count%;
Profile("My data", "path", "c:\\work", path$); 'get initial path
Profile("My data", "count", 0, count%); 'and initial count
...                                     'your script...
Profile("My data","path", path$);       'save final value
Profile("My data","count", count%);     'save final count
```

Registry use by Signal

The `HKEY_CURRENT_USER\Software\CED\Signal` key contains the following keys that are used by Signal:

BarList This key holds the list of scripts to load into the script bar when Signal starts.

Editor This key holds the editor settings for scripts, output sequences and general text editing.

PageSetup This key holds the margins in units of 0.01 mm for printing data views, and the margins in mm for text-based views. It also holds header and footer text for text-based views.

Preferences The values in this key are mainly set by the Edit menu preferences. If you change any Edit menu Preferences value in this key, Signal will use the changed information immediately. The values are all integers except the file path, which is a string:

Assume Power	0=Do not assume Power1401 hardware, 1=do assume.
Differ Optimise	0=Y-axis optimised on data aquired so far when requested on-line, 1=Y-axis optimise differed to sweep end when requested on-line.
Enhanced Metafile	0=Windows metafile, 1=enhanced metafile for clipboard.
Enter debug on error	1=Enter the script debugger when an error occurs. 0=Do not.
File shorts	0=Waveform data to be written to CFS file as 16 bit integers, 1=to be written as floating point values. Add 256 if calibrated zero is to be kept at zero volts.
File update	0=Discard changes to data, 1=query the user, 2=always save changes.
Font Italic	0=Use non-italic font as default for file and memory views. 1=use italic font.

Font Name	Name of font to use as default for file and memory views.
Font Pitch	0=Use default-pitch font for the above, 1=use a fixed-pitch font (all characters the same width), 2=use a variable-pitch font. To this value you should add: 4=don't care which family of font used, 8=use a serifed, variable-width font, 16=sans-serif, variable width font, 32=constant-width font, 64=cursive font, 128=decorative font.
Font Size	Size of font to use as default for file and memory views.
Font Weight	400=Normal font, 700=bold font.
Force idle cycles	0 – 65535 number of time a script idle is called before handing time back to the System. 0=No limit.
Force idle time	Number of ms to force Signal to idle for before giving time back to System (0-200). 0=No limit.
Frame Time mode	0=Seconds, 1=HH:MM:SS, 2=Time of day.
Line thickness codes	Bits 0-3 = Axis code, bits 4-7 = Data code. The codes 0-15 map onto the 16 values in the drop down list. Bit 7=1 to use lines, not rectangles, to draw axes.
Low channels at top	0=Standard display shows low channel at bottom, 1=at top.
Metafile Scale	0-11 selects from the list of allowed scale factors.
New file path	New data file directory or blank for current folder.
No flicker free drawing	0=Use flicker free drawing, 1=Do not.
No save prompt	0=Prompt to save derived views, 1=no prompt.
Online ADC range	0=Maintain always, 1=Keep showing full ADC range if already doing so, 2=Maintain ADC range percentage.
Prompt comment	1=Prompt for File Comment when sampling stops. 0=Do not.
Save modified scripts	1=Save modified scripts before running them. 0=Do not save.
Time mode	0=Display seconds, 1=display ms, 2=display µs.
Update interval	Number of ms between on-line updates or script idles.
Provide clamp features	0=No clamping, 1=clamping.

The keys with names starting `"Bars-"` are used by system code to restore dockable toolbars. You can delete them all safely; any other change is likely to crash Signal.

Recent file list This key holds the list of recently used files that appear at the bottom of the file menu.

Recover This key holds the information to recover data from interrupted sampling sessions.

Settings This is where the evaluate bar saves the last few evaluated lines.

Tip The *Tip of the Day* dialog uses this key to remember the last tip position.

Version Signal uses this key to detect when a new version of the program is run for the first time.

Win32 In Windows NT derived systems, this key holds the desired working set sizes. The working set sizes in use are displayed in the Help menu About Signal dialog. Click the Help button in this dialog to read more about using these registry values. The values are as follows:

Minimum working set	Minimum size in kB (units of 1024 bytes), default is 800.
Maximum working set	Maximum size in kB, default is 4000 (4 MB).

See also: `ViewUseColour()`

ProgKill()

This function terminates a program started using `ProgRun()`. This is a very powerful function. It will terminate a program without giving it the opportunity to save data. Use this command with care!

```
Func ProgKill(pHdl%);
```

`pHdl%` A program handle returned by `ProgRun()`.

Returns Zero or a negative error code.

See also: `ProgRun()`, `ProgStatus()`

ProgRun()

This function runs a program using command line arguments as if from a DOS-style command prompt.

```
Func ProgRun(cmd$ {,code% {,xLow, yLow, xHigh, yHigh}});
```

`cmd$` The command string as would be typed at a DOS-style prompt.

`code%` If present, this sets the initial application window state: 0=Hidden, 1=Normal, 2=Iconised, 3=Maximised. Some programs set their own window sizes and styles so this may not work. The next 4 arguments set the Normal window position:

`xLow` Position of the left window edge as a percentage of the screen width.

`yLow` Position of the top window edge as a percentage of the screen height.

`xHigh` The right hand edge as a percentage of the screen width.

`yHigh` The bottom edge position as a percentage of the screen height.

Returns A program handle or a negative error code.

See also: `ProgKill()`, `ProgStatus()`, `System$()`

ProgStatus()

This function is used to check if a program started with `ProgRun()` is still running. If it finds that the program has terminated it will close the handle which will then become invalid if used again.

```
Func ProgStatus(pHdl%);
```

`pHdl%` The program handle returned by `ProgRun()`.

Returns 1 if the program is still running, 0 if it has terminated or a negative error code.

See also: `ProgKill()`, `ProgRun()`

ProtocolAdd()

This function adds a new protocol to the list of protocols defined in the sampling configuration. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling.

```
Func ProtocolAdd(name$);
```

`name$` The name for the new protocol, which must not be blank.

Returns The number for the new protocol or a negative error code.

See also: `Protocols()`, `ProtocolDel()`, `ProtocolClear()`, `ProtocolName$()`

ProtocolClear()

This function initialises a protocol defined in the sampling configuration. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling.

```
Func ProtocolClear(num%|name$);
```

`num%` The number of the protocol to clear, from 1 to the number returned by `Protocols()`.

`name$` The name of the protocol to be cleared.

Returns Zero or a negative error code.

See also: `Protocols()`, `ProtocolDel()`, `ProtocolAdd()`, `ProtocolName$()`

ProtocolDel()

This function deletes a protocol from the list defined in the sampling configuration. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling.

```
Func ProtocolDel(num%|name$);
```

`num%` The number of the protocol to delete, from 1 to the number returned by `Protocols()`.

`name$` The name of the protocol to be deleted.

Returns Zero or a negative error code.

See also: `Protocols(), ProtocolAdd(), ProtocolClear(), ProtocolName$()`

ProtocolEnd()

This sets what happens when the end of a protocol is reached. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling.

```
Func ProtocolEnd(num%|name${, atEnd%});
```

`num%` The number of the protocol to delete, from 1 to the number returned by `Protocols()`.

`name$` The name of the protocol to be deleted.

`atEnd%` Set to 0 for the protocol finishing, or 1 to n to select chaining to protocol 1 to n. No protocol is selected if `next` value is above current count of protocols.

Returns The previous value of `atEnd%`.

See also: `Protocols(), ProtocolAdd(), ProtocolClear(), ProtocolName$()`

ProtocolFlags()

This function gets the flags for a protocol defined in the sampling configuration and optionally sets the flags to a new value. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling.

```
Func ProtocolFlags(num%|name$ {, new%});
```

`num%` The number of the protocol to use, from 1 to the number returned by `Protocols()`.

`name$` The name of the protocol to use.

`new%` If present, the new protocol flags value. This is the sum of values for each flag option, the values are:

1	Initialise pulse variations when protocol starts.
2	Sampling switches to state 0 when protocol finishes.
4	Turn on writing to disk when protocol starts.
8	Selects not turning off writing to disk on finishing.
16	Selects creation of a button in the control bar for this protocol.
32	Selects cycling protocol states only on writing, otherwise always.

Returns The flags for this protocol before the call.

See also: `Protocols(), ProtocolClear(), ProtocolName$(), ProtocolRepeats()`

ProtocolName$()

This function gets the name for a protocol defined in the sampling configuration and optionally sets a new name. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling.

```
Func ProtocolName$(num%|name$ {, new$});
```

`num%` The protocol number to use, from 1 to the number returned by `Protocols()`.

`name$` The name of the protocol to use.

`new$` If present, this sets the new protocol name.

Returns The name of the protocol before this call.

See also: `Protocols(), ProtocolAdd(), ProtocolDel(), ProtocolClear()`

ProtocolRepeats()

This function gets the name for a protocol defined in the sampling configuration and optionally sets a new name. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling.

```
Func ProtocolRepeats(num%|name$ {, new%});
```

`num%` The protocol number to use, from 1 to the number returned by `Protocols()`.

`name$` The name of the protocol to use.

`new%` If present, sets the new repeat count. Any repeat count from 1 to 1000 can be set, or zero for a protocol which repeats forever.

Returns The repeat count for the protocol before the call.

See also: `Protocols(), ProtocolFlags(), ProtocolEnd()`

Protocols()

This function gets the number of protocols defined in the sampling configuration. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling.

```
Func Protocols();
```

Returns The number of protocols defined in the sampling configuration.

See also: `ProtocolAdd(), ProtocolDel(), ProtocolClear(), ProtocolName$()`

ProtocolStepGet()

This function gets information about a specific step within a protocol from the list defined in the sampling configuration. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling.

```
Func ProtocolStepGet(num%|name$, step, &state, &repeat, &next);
```

`num%` The protocol number to use, from 1 to the number returned by `Protocols()`.

`name$` The name of the protocol to use.

`step` The step in the protocol, from 1 to 10.

`state` This parameter is updated with the state for this step.

`repeat` This parameter is updated with the repeat count for this step.

`next` This parameter is updated with the next step value for this step.

Returns Zero or a negative error code.

See also: `Protocols(), ProtocolFlags(), SampleStates(), ProtocolClear(), ProtocolName$()`

ProtocolStepSet()

This function sets up a specific step within a protocol from the list defined in the sampling configuration. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling.

```
Func ProtocolStepSet(num%|name$, step%, state, repeat, next);
```

`num%` The number of the protocol to use, from 1 to the number returned by `Protocols()`.

`name$` The name of the protocol to use.

`Step%` The step in the protocol, from 1 to 10.

`state` This parameter sets the state for this step, from 0 to 256.

`repeat` This parameter sets the repeat count for this step, from 1 to 1000.

`next` This parameter sets the next step value for this step, from 0 to 10. A value of zero terminates protocol execution.

Returns Zero or a negative error code.

See also: `Protocols()`, `ProtocolFlags()`, `SampleStates()`, `ProtocolClear()`, `ProtocolName$()`

PulseXXX() Pulse output commands

The PulseXXX family of commands can be used to control the pulse outputs generated during sampling sweeps. Pulses can be generated on up to eight 1401 DACs and on 8 bits of dedicated digital output. For the micro1401 and Micro1401 mk II, only two DACs are available. These functions normally operate on the stored sampling configuration but if used during sampling they operate upon the on-going sampling.

As part of the Signal multiple states facilities, each state can have a separate set of pulse outputs. Because of this, all script functions that access the pulses information have a parameter to select the state. For single states, set this parameter to zero.

Individual pulses can be specified by their number or by name. For access by number the pulses for a given output are kept in a sorted list in order of their start time. The (always present) initial level is zero, subsequent pulses are 1 and upwards. Though access by number seems straightforward, it does have some drawbacks. Firstly, when the start time of a pulse is changed the ordering of the list can change and the pulse number will be changed. Secondly, for complex reasons, the arbitrary waveform output item is always attached to the DAC 0 outputs list, regardless of what DACs it uses, and does not appear in any other output lists. This can make things very confusing! Therefore we recommend that, for non-trivial pulse output arrangements, individual pulses are accessed by name.

See also: `PulseAdd(), PulseDataGet(), PulseDataSet(), PulseDel(), PulseFlags(), PulseName$(), Pulses(), PulseTimesGet(), PulseTimesSet(), PulseType(), PulseVarGet(), PulseVarSet(), PulseWaveformGet(), PulseWaveformSet(), PulseWaveGet(), PulseWaveSet()`

PulseAdd()

This function adds a new pulse to the output pulses for a given state and output. If this is used to add an arbitrary waveform output to pulses that already contain such an item, the pre-existing waveform output is deleted.

```
Func PulseAdd(state%, out%, type%, name$, time, len {, flags%});
```

`state%` The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`out%` The output to which this applies. Values of zero upwards select the corresponding DAC output, -1 selects the digital outputs.

`type%` A code for the type of pulse. The legal codes are:

1 A simple square pulse.
2 A square pulse with varying amplitude.
3 A square pulse with varying duration.
4 A ramp pulse.
5 A cosine wave segment.
6 An arbitrary waveform; the rate is initialised to 100Hz.
7 A pulse train.

`name$` The name for the new pulse. This can be blank.

`time` The start time for the pulse in seconds from the start of the outputs.

`len` The length of the pulse, in seconds. For a pulse train, this is the length of the individual pulses in the train, not the length of the entire train.

`flags%` If present, this sets the flags for the pulse. Flag bit 0 is set for varying-width pulses to push following pulses back, bit 1 is set for pulses to stay up at the end. If this parameter is omitted, the pulse flags are cleared.

Returns The number of the new pulse or a negative error code. The initial level item is always present as pulse zero, so the smallest successful return value is 1.

See also: `Pulses(), PulseDel(), SampleStates(), SampleOutLength(), PulseName$()`

PulseClear()

This function deletes all the pulses for a given state and output and sets the initial level to zero.

```
Func PulseClear(state%, out%);
```

`state%` The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`out%` The output to which this applies. Values of zero upwards select the corresponding DAC output, -1 selects the digital outputs.

Returns Zero or a negaitive error code.

See also: `Pulses(), PulseAdd(), PulseDel(), SampleStates()`

PulseDataGet()

This function retrieves the amplitude and other values for a pulse in the outputs for a given state and output. Up to four data values can be retrieved, the meaning of most of these varies with the pulse type. A separate function, `PulseWaveGet()`, retrieves the settings for waveform outputs.

```
Func PulseDataGet(state%, out%, num%|name$, &amp {, &val1 {, &val2
                    {, &val3}}});
```

`state%` The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`out%` The output to which this applies. Values of zero upwards select the corresponding DAC output, -1 selects the digital outputs.

`num%` The number of the pulse in question, from 0 to the number of pulses - 1.

`name$` The name of the pulse to use.

`amp` This is updated with the amplitude or level of the pulse, or the bit value for digital pulses.

`val1` This is updated with the end amplitude for ramps, the initial phase for cosines, and the number of pulses for pulse trains.

`val2` This is updated with the step mode for ramps, the centre value for sines and the gap for pulse trains. The step mode value is 0 for both ends, 1 for start only and 2 for end only.

`val3` This is updated with the cycle period for sines only.

Returns Zero or a negative error code.

See also: `Pulses(), PulseAdd(), PulseDataSet(), PulseName$()`

PulseDataSet()

This function sets the amplitude and other values for a pulse in the outputs for a given state and output. Up to four data values can be set, the meaning of most of these varies with the pulse type. A separate function, `PulseWaveSet()`, is used to change the settings for arbitrary waveform output.

```
Func PulseDataSet(state%, out%, num%|name$, amp {, val1 {, val2
                      {, val3}}}});
```

`state%` The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`out%` The output to which this applies. Values of zero upwards select the corresponding DAC output, -1 selects the digital outputs.

`num%` The number of the pulse in question, from 0 to the number of pulses - 1.

`name$` The name of the pulse to use.

`amp` This sets the amplitude or level of the pulse, or the bit value for digital pulses.

`val1` This sets the end amplitude for ramps, the initial phase for cosines, and the number of pulses for pulse trains.

`val2` This sets the step mode for ramps, the centre value for sines and the gap between pulses for pulse trains. The step mode value is 0 for both ends, 1 for start only and 2 for end only.

`val3` This sets the cycle period for sines only.

Returns Zero or a negative error code.

See also: `Pulses(), PulseAdd(), PulseDataGet(), PulseName$()`

PulseDel()

This function deletes a pulse from the output pulses for a given state and output.

```
Func PulseDel(state%, out%, num%|name$);
```

`state%` The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`out%` The output to which this applies. Values of zero upwards select the corresponding DAC output, -1 selects the digital outputs.

`num%` The number of the pulse to delete, from 1 to the number of pulses - 1 (you cannot delete pulse zero; the initial level).

`name$` The name of the pulse to delete. You cannot delete the initial level.

Returns Zero or a negative error code.

See also: `Pulses(), PulseAdd(), SampleStates(), SampleOutMode()`

PulseFlags()

This function retrieves, and optionally sets, the options flags for a pulse in the outputs for a given state and output.

```
Func PulseFlags(state%, out%, num%|name$ {, flags});
```

`state%` The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`out%` The output to which this applies. Values of zero upwards select the corresponding DAC output, -1 selects the digital outputs.

`num%` The number of the pulse in question, from 0 to the number of pulses - 1.

`name$` The current name of the pulse in question.

`flags` If present, this sets the new flags for the pulse. Flag bit 0 is set for varying-width pulses to push following pulses back, bit 1 is set for pulses to stay up at the end.

Returns The flags for the pulse at the time of the function call.

See also: `Pulses(), PulseAdd(), SampleStates(), SampleOutMode()`

PulseName$()

This function retrieves, and optionally sets, the name of a pulse in the output pulses for a given state and output.

```
Func PulseName$(state%, out%, num%|name$ {, new$});
```

`state%` The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`out%` The output to which this applies. Values of zero upwards select the corresponding DAC output, -1 selects the digital outputs.

`num%` The number of the pulse in question, from 0 to the number of pulses - 1.

`name$` The current name of the pulse in question.

`new$` The new name for the pulse. Blank pulse names are legal.

Returns The name of the pulse at the time of the function call.

See also: `Pulses(), PulseAdd(), SampleStates(), SampleOutMode()`

Pulses()

This function returns the number of pulses for a given state and output. This number is usually straightforward to use, but can be complicated by the fact that waveform output items always appear on the DAC 0 outputs, regardless of the DACs in use, and not on any other DAC.

```
Func Pulses(state%, out%);
```

`state%` The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`out%` The output to which this applies. Values from 0 upwards select the corresponding DAC, -1 selects the digital outputs.

Returns The number of pulses on this output. This value will be 1 or more as there is always one pulse defined for an output, the initial level.

See also: `PulseDel(), PulseAdd(), PulseWaveSet(), PulseName$()`

PulseTimesGet()

This function retrieves the times for a pulse in the outputs for a given state and output.

```
Func PulseTimesGet(state%, out%, num%|name$, &time, &len);
```

`state%` The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`out%` The output to which this applies. Values of zero upwards select the corresponding DAC output, -1 selects the digital outputs.

`num%` The number of the pulse in question, from 1 to the number of pulses - 1. Zero is not meaningful because there are no times for the initial level.

`name$` The name of the pulse to use.

`time` This is updated with the start time for the pulse, in seconds from the start of the pulse outputs.

`len` This is updated with the length of the pulse, in seconds.

Returns Zero or a negative error code.

See also: `Pulses(), PulseAdd(), PulseTimesSet(), SampleOutMode()`

PulseTimesSet()

This function sets the times for a pulse in the outputs for a given state and output.

```
Func PulseTimesSet(state%, out%, num%|name$, time, len);
```

`state%` The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`out%` The output to which this applies. Values of zero upwards select the corresponding DAC output, -1 selects the digital outputs.

`num%` The number of the pulse in question, from 1 to the number of pulses - 1. Zero is not usable because there are no times for the initial level.

`name$` The name of the pulse to use.

`time` This sets the start time for the pulse, in seconds from the start of the pulse outputs.

`len` This sets the length of the pulse in seconds. This does not affect arbitrary waveform items; use `PulseWaveSet()`.

Returns Zero or a negative error code.

See also: `Pulses(), PulseAdd(), PulseTimesGet(), SampleOutMode()`

PulseType()

This function returns a code for the type of a pulse in outputs for a given state and output.

```
Func PulseType(state%, out%, num%|name$);
```

`state%` The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`out%` The output to which this applies. Values of zero upwards select the corresponding DAC output, -1 selects the digital outputs.

`num%` The number of the pulse in question, from 0 to the number of pulses - 1.

`name$` The name of the pulse in question.

Returns A code for the type of pulse, as follows:

0 The initial level for the output.
1 A simple square pulse.
2 A square pulse with varying amplitude.
3 A square pulse with varying duration.
4 A ramp pulse.
5 A cosine wave segment.
6 An arbitrary waveform.
7 A pulse train.

See also: `Pulses(), PulseAdd(), PulseDataGet(), PulseDataSet()`

PulseVarGet()

This function retrieves the values controlling the automatic variation of a pulse in the outputs for a given state and output.

```
Func PulseDataGet(state%, out%, num%|name$, &step{, &repeat
                    {, &steps}});
```

`state%` The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`out%` The output to which this applies. Values of zero upwards select the corresponding DAC output, -1 selects the digital outputs.

`num%` The number of the pulse in question, from 0 to the number of pulses - 1.

`name$` The name of the pulse in question.

`step` This is updated with the step value for the pulse.

`repeat` This, if present, is updated with the repeat count for each step in the variation.

`steps` This, if present, is updated with the total steps for the variation. Note that the number of values for the variation is `steps+1` as the initial value is also used.

Returns Zero or a negative error code.

See also: `Pulses(), PulseAdd(), PulseVarSet(), PulseName$()`

PulseVarSet()

This function sets values controlling the automatic variation of a pulse in the outputs for a given state and output.

```
Func PulseVarSet(state%, out%, num%|name$, step{, repeat
                        {, steps}});
```

`state%` The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`out%` The output to which this applies. Values of zero upwards select the corresponding DAC output, -1 selects the digital outputs.

`num%` The number of the pulse in question, from 0 to the number of pulses - 1.

`name$` The name of the pulse to use.

`step` This sets the step value for the pulse.

`repeat` This, if present, sets the repeat count for each step in the variation.

`steps` This, if present, sets the total steps for the variation. Note that the number of values for the variation is `steps+1` as the initial value is also used.

Returns Zero or a negative error code.

See also: `Pulses(), PulseAdd(), PulseVarGet(), PulseName$()`

PulseWaveformGet()

This function retrieves the waveform data values sent to a given DAC as part of the arbitrary waveform item in the pulses for a given state.

```
Func PulseWaveformGet(state%, dac, dat%[]|dat[]);
```

`state%` The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`dac` The DAC number for which we want data. If the DAC is not used, no data will be returned.

`dat%[]` An integer array that will be filled with the data for the DAC. The values are the 16-bit integer values that would be written to the DAC. A value of -32768 corresponds to an output of -5 volts, 32767 corresponds to +5 volts. If the array is too short to hold all of the waveform, it will be filled. If the array is longer than the waveform, all the points will be copied and the rest of the array left unchanged.

`dat[]` A real array that will be filled with the data for the DAC. The values are calibrated using the appropriate DAC scaling factors. If the array is too short to hold all of the waveform, it will be filled. If the array is longer than the waveform, all the points will be copied and the rest of the array left unchanged.

Returns The number of points copied or a negative error code.

See also: `Pulses(), PulseAdd(), PulseWaveSet(), PulseWaveGet(), PulseWaveformSet()`

PulseWaveformSet()

This function sets the waveform data values sent to a given DAC as part of the arbitrary waveform item in the pulses for a given state.

`Func PulseWaveformSet(state%, dac, dat%[]|dat[]);`

`state%` The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`dac` The DAC number for the data. If the DAC is not used, the function will do nothing.

`dat%[]` An integer array that holds the new data for the DAC. The values are the 16-bit integer values that would be written to the DAC. A value of -32768 corresponds to an output of -5 volts, 32767 corresponds to +5 volts. If the array is too short to hold all the waveform points, only the earlier points in the waveform will be changed. If the array is longer than the waveform, the extra data in the array is unused.

`dat[]` A real array that holds the new data for the DAC. The values are converted into DAC output values using the appropriate DAC scaling factors. If the array is too short to hold all of the waveform points, only the earlier points in the waveform will be changed. If the array is longer than the waveform, the extra data in the array is unused.

Returns The number of waveform points changed or a negative error code.

See also: `Pulses(), PulseAdd(), PulseWaveSet(), PulseWaveGet(), PulseWaveformGet()`

PulseWaveGet()

This function retrieves the values controlling the arbitrary waveform item in the pulses for a given state.

`Func PulseWaveGet(state%, &mask, &rate, &points);`

`state%` The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`mask` This is updated with the DAC mask value for the output. This has one bit set for each DAC used, bit 0 for DAC 0 and so forth.

`rate` This is updated with the output rate for the waveform data, in Hz.

`points` This is updated with the number of points of data for each DAC.

Returns Zero or a negative error code.

See also: `Pulses(), PulseAdd(), PulseWaveSet()`

PulseWaveSet()

This function sets the values controlling the arbitrary waveform item in the pulses for a given state.

`Func PulseWaveSet(state%, mask, rate, points);`

`state%` The state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`mask` This sets the DAC mask value for the output, which controls which DACs are used. This has one bit set for each DAC used, bit 0 for DAC 0 and so forth.

`rate` This sets the output rate for the waveform data, in Hz.

`points` This sets the number of points of data for each DAC in the mask.

Returns Zero or a negative error code.

See also: `Pulses(), PulseAdd(), PulseWaveGet()`

Query()

This function is used to ask the user a Yes/No question. It opens a window with a message and two buttons. The window is removed when a button is pressed.

```
Func Query(text1$, {,Yes$ {,No$}});
```

`text1$` This string forms the text in the window. If the string contains a vertical bar character (|), the portion of the string before the | will be used to set the title of the window. Up to 20 lines of text can be shown, each up to 60 characters long.

`Yes$` This sets the text for the first button. If this argument is omitted, `"Yes"` is used.

`No$` This sets the text for the second button. If this is omitted, `"No"` is used.

Returns 1 if the user selects Yes or presses Enter, 0 if the user selects the No button.

See also: `Print(), Input(), Message(), DlgCreate()`

Rand()

This function returns pseudo-random numbers with a uniform density over a given range. The values returned are ***R*** * `scl` + `off` where ***R*** is a random number in the range 0 up to, but not including, 1. When you start Signal, the generator is initialised with a random seed based on the time. If you require a repeatable sequence, you must set the seed. The sequence is independent of `RandExp()` and `RandNorm()`.

```
Func Rand(seed);
Func Rand({scl, off});
Func Rand(arr[]{[]}{, scl{, off}});
```

`seed` If present, this is a seed for the generator in the range 0 to 1. If `seed` is outside this range, the fractional part of the number is used. If you use 0.0 as the seed, the generator is initialised with a seed based on the time.

`arr` This 1 or 2 dimensional real or integer array is filled with random numbers. If an integer array is used, the random number is truncated to an integer.

`scl` This scales the random number. If omitted, it has the value 1.

`off` This offsets the random number. If omitted it has the value 0.

Returns If the first argument is not an array, the return value is a random number in the range `off` up to `off+scl`. If a seed is given, a random number is still returned. If the first argument is an array, the return value is 0.0.

See also: `RandExp(), RandNorm()`

RandExp()

This function returns pseudo-random numbers with an exponential density, suitable for generating Poisson statistics. The values returned are ***R*** * `mean` + `off` where ***R*** is a random number with the density function $p(x) = e^{-x}$. When you start Signal, the generator is initialised with a random seed based on the time. For repeatable sequences, you must set a seed. The sequence is independent of `Rand()` and `RandNorm()`.

```
Func RandExp(seed);
Func RandExp({mean, off});
Func RandExp(arr[]{[]}{, mean{, off}});
```

`seed` If present, this is a seed for the generator in the range 0 to 1. If `seed` is outside this range, the fractional part of the number is used. If you use 0.0 as the seed, the generator is initialised with a seed based on the time.

`arr` This 1 or 2 dimensional real or integer array is filled with random numbers. If an integer array is used, the random number is truncated to an integer.

`mean` This scales the random number. If omitted, it has the value 1.

`off` This offsets the random number. If omitted it has the value 0.

Returns If the first argument is not an array, the return value is a random number. If a seed is given, a random number is still returned. If the first argument is an array, the return value is 0.0.

The following example fills an array with event times with a mean interval `t`:

```
RandExp(arr[], t);          'Fill arr with event intervals
ArrIntgl(arr[]);            'convert intervals to times
```

See also: `Rand()`, `RandNorm()`

RandNorm()

This function returns pseudo-random numbers with a normal density. The values returned are ***R*** * `scl` + `off` where ***R*** is a random number with a normal probability density function $p(x) = (2\pi)^{-½}e^{-x^2/2}$; this has a mean of 0 and a variance of 1. When you start Signal, the generator is initialised with a random seed based on the time. For a repeatable sequence, you must set a seed. The sequence is independent of `Rand()` and `RandExp()`.

```
Func RandNorm(seed);
Func RandNorm({scl, off});
Func RandNorm(arr[]{[]}{, scl{, off}});
```

`seed` If present, this is a seed for the generator in the range 0 to 1. If `seed` is outside this range, the fractional part of the number is used. If you use 0.0 as the seed, the generator is initialised with a seed based on the time.

`arr` This 1 or 2 dimensional real or integer array is filled with random numbers. If an integer array is used, the random number is truncated to an integer.

`scl` This scales the random number. If omitted, it has the value 1.

`off` This offsets the random number. If omitted it has the value 0.

Returns If the first argument is not an array, the return value is a random number. If a seed is given, a random number is still returned. If the first argument is an array, the return value is 0.0.

See also: `Rand()`, `RandExp()`

Read()

This function takes the next line read from the current view as the source of a text string and converts the text into variables. The read starts at the current position of the text cursor. The text cursor moves to the start of the next line after the read.

```
Func Read({&var1 {, &var2 {,&var3 ...}}});
```

`varn` Arguments must be variables. They can be any type. One dimensional arrays are allowed. The variable type determines how the function extracts data from the string. In a successful call, each variable will be matched with a field in the string, and the value of the variable is changed to the value found in the field.

A call to `Read()` with no arguments skips a line.

Returns The function returns the number of fields in the text string that were successfully extracted and returned in variables, or a negative error code. Attempts to read past the end of the file produce the end of file error code.

It is not considered an error to run out of data before all the variables have been updated. If this is a possibility you must check that the number of items returned matches the number you expected. If an array is passed in, it is treated as though it was the number of individual values held in the array.

The source string is expected to hold data values as real numbers, integer numbers and strings. Strings can be delimited by quote marks, for example `"This is a string"`, or they can be just text. However, if a string is not delimited by quotes, it is deemed to run to the end of the source string, so no other items can follow it.

The fields in the source string are separated by white space (tabs and spaces) and commas. Space characters are "soft" separators. You can have any number of spaces between fields. Tabs and commas are treated as "hard" separators. Two consecutive tabs or commas, or a tab and a comma (with or without intervening spaces), imply a blank field. When reading a field, the following rules are applied:

1. Space characters are skipped over
2. Characters that are legal for the variable into which data is to be read are extracted until a non-legal character or a separator or end of data is found. The characters read are converted into the variable type. If an error occurs in the translation, the function returns the error. Blank fields assigned to numbers are treated as 0. Blank fields assigned to strings produce empty strings.
3. Characters are then skipped until a separator character is found or the end of the data is reached. If the separator is a space, it and any further spaces are skipped. If the next character is a hard separator it is also skipped.
4. If there are no more variables or no more data, the process stops, or else goes back to step 1.

Example The following example shows a source line, followed by a `Read()` function, then the assignment statements that would be equivalent to the `Read()`:

`"This is text"    ,  2 3 4,, 4.56 Text too 3 4 5` The source line

```
n := Read(fred$, jim[1:2], sam, dick%, tom%, sally$, a, b, c);
n := 7;
fred$ := "This is text";
jim[1] := 2; jim[2] := 3;
sam := 4;
dick% := 0;
tom% := 4;
sally$ := "Text too 3 4 5"
a, b and c are not changed
```

See also: `FileOpen(), ReadStr(), ReadSetup()`

ReadSetup()

This sets the separators and delimiters used by `Read()` and `ReadStr()` to convert text into numbers and strings. You can also set string delimiters and set a string separator.

```
Proc ReadSetup({hard${, soft${, sDel${, eDel${, sSep$}}}}});
```

`hard$` The characters to use as hard separators between all fields. If this is omitted or the string is empty, the standard hard separators of comma and tab are used.

`soft$` The characters to use as soft separators. If this is omitted, the space character is set as a soft separator. If `soft$` is empty, no soft separators are used.

`sDel$` The characters that delimit the start of a string. If omitted, a double quote is used. If empty, no delimiter is set. Delimiters are not returned in the string.

`eDel$` The characters that delimit the end of a string. If omitted, a double quote is used. If empty, no delimiter is set. If `sDel$` and `eDel$` are the same length, only the end delimiter character that matches the start delimiter position is used. For example, to delimit strings with <text> or 'text' set sDel$ to "<'" and eDel$ to ">'". You can repeat a character to force different lengths.

`sSep$` The list of hard separator characters for strings that have no start delimiter. For example, setting "|" lets you read one|two|three into three separate strings.

See also: `Read(), ReadStr(), Val()`

ReadStr()

This function extracts data fields from a string and converts them into variables.

```
Func ReadStr(text$, &var1 {, &var2 {, &var3...}});
```

`text$` The string used as a source of data.

`var` The arguments must all be variables. The variables can be of any type, and can be one dimensional arrays. The type of each variable determines how the function tries to extract data from the string. See `Read()` for details.

Returns The function returns the number of fields in the text string that were successfully extracted and returned in variables, or a negative error code.

It is not an error to run out of data before all the variables have been updated. If this is a possibility you must check the returned value. If an array is passed in, it is treated as though it was the number of individual values held in the array.

See also: `Read(), Print$(), Val(), ReadSetup()`

Right$()

This function returns the rightmost `n` characters of a string.

```
Func Right$(text$, n);
```

`text$` A string of text.

`n` The number of characters to return.

Returns The last `n` characters of the string, or all the string if it is less than `n` characters.

See also: `DelStr$(), InStr(), Left$(), Len(), Mid$()`

Round()

Rounds a real number or an array of reals to the nearest whole number.

```
Func Round(x|x[]{[]...});
```

`x` A real number or an array of reals.

Returns If x is an array it returns 0. Otherwise it returns a real number with no fractional part that is the nearest to the original number.

See also: `Trunc()`, `Frac()`

SampleAbort()

This cancels sampling and deletes the views that were being sampled and any associated result and cursor views. If a memory view derived from the data has been saved, the saved file remains. It is equivalent to the Abort button on the floating sampling control window.

```
Func SampleAbort();
```

Returns 0 if sampling was aborted, or a negative error code.

See also: `SampleReset()`, `SampleStart()`, `SampleStop()`, `SampleStatus()`

SampleAbsLevel()

This function sets and gets the absolute levels flag as seen in the sampling configuration dialog Outputs page.

```
Func SampleAbsLevel({new});
```

`new` If present, this sets the new absolute levels flag if non-zero, clears the flag if zero.

Returns The absolute levels flag from the configuration at the time of the call.

See also: `SampleClear()`, `Pulses()`

SampleAccept()

This tags the current frame as accepted or rejected.

```
Func SampleAccept({yes%});
```

`yes%` If this is zero, the frame is rejected, otherwise the frame is written to disk. This is equivalent to the Accept check box in the sampling control dialog.

Returns 0 if sweep was written successfully or a negative error code.

See also: `SamplePause()`, `SampleStatus()`, `SampleWrite()`

SampleArtefactGet()

This command returns the parameters used in automatic artefact rejection during sampling. Artefact rejection consists of testing all points within a given time range and rejecting or tagging frames where the points at the ADC limit exceed a set threshold. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling.

```
Func SampleArtefactGet(mode%, start, end, per{, lev});
```

`mode%` Returned holding the artefact rejection mode: 0 for none, 1 for Tag and 2 for Reject.

`start` Returned holding the start time for the search for out-of-range points.

`end` Returned holding the end time for the search for out-of-range points.

`per` Returned holding the percentage of out-of-range points that can be tolerated, frames with more than this are deemed to contain artefacts.

`lev` The percentage of the ADC range (from zero to full scale) above or below which will be considered an artefact.

Returns Zero or a negative error code.

See also: `SampleArtefactSet()`, `SampleAccept()`, `FrameTag()`, `SamplePortFull()`

SampleArtefactSet()

This command sets the parameters used in automatic artefact rejection during sampling. Artefact rejection consists of testing all points within a given time range and rejecting or tagging frames where the points at the ADC limit exceed a set threshold. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling.

```
Func SampleArtefactSet(mode%, start, end, per{, lev});
```

`mode%` Sets the artefact rejection mode: 0 for none, 1 for Tag and 2 for Reject.

`start` The start time for the search for out-of-range points.

`end` The end time for the search for out-of-range points.

`per` The percentage of out-of-range points that can be tolerated, frames with more than this are deemed to contain artefacts.

`lev` The percentage of the ADC range (from zero to full scale) above or below which will be considered an artefact. The default is 100.

Returns Zero or a negative error code.

See also: `SampleArtefactGet()`, `SampleAccept()`, `FrameTag()`, `SamplePortFull()`

SampleAutoFile()

This gets or sets the flag for file auto-filing as seen in the sampling configuration dialog.

```
Func SampleAutoFile({yes%});
```

`yes%` If present and non-zero, this turns on automatic filing of data when sampling finishes. If zero or missing it turns automatic filing off.

Returns The automatic filing flag at the time of the function call.

See also: `SampleAutoName$()`, `FileNew()`

SampleAutoName$()

This gets or sets the template for file auto-naming as in the sampling configuration dialog.

```
Func SampleAutoName$({name$});
```

`name$` If present, this sets the new template string for file auto-naming, or turns off auto-naming if it is a blank string. See the sampling configuration documentation for details on the template string.

Returns The auto-naming template at the time of the function call.

See also: `SampleAutoFile(), FileNew()`

SampleAuxStateParam()

This function is used to get or set parameters for the auxiliary states device, if one is installed. The meaning and use of these parameters varies according to the type of auxiliary states device in use. At the time of writing the auxiliary devices supported are the MagStim and the CED 3304. The meaning of the `num%` parameter in this function will depend entirely upon the type of auxiliary states device installed. It is therefore very important that, before using these functions, the value returned by `SampleAuxStateParam(0)` is checked, as well as for a MagStim calling `SampleAuxStateParam(1)` to check its type and that nothing is done by the script if the returned values do not match the supported devices.

Warning Both the MagStim and CED 3304 are potentially dangerous devices. If you use these script functions carelessly, they could be set to deliver high levels of stimulation or magnetic field. CED cannot accept any responsibility whatsoever for any harm or damage resulting.

```
Func SampleAuxStateParam(num%, {val});
```

`num%` Selects the parameter to read or set. The meaning of all parameters except 0 depends on the auxiliary device type.

0 The type of auxiliary device support loaded. -1 = no device, 1=Magstim, 2=CED 3304 current stimulator. You cannot set this parameter.

For a MagStim only:

1 The device in use. -1 = Do not use, 0 = 200^2, 1 = $BiStim^2$, 2 = $Rapid^2$, 3 = dual 200^2.

2 The device flags. This is the sum of 1 for the $Rapid^2$ coil interlock ignore, 2 for $BiStim^2$ high-resolution timing, 4 for $Rapid^2$ single pulse mode and 8 for assume $BiStim^2$ independent triggers. Not all of these values are meaningful with any particular MagStim type; your script code needs to be carefully written.

3 The serial line port (COM port) used to control the MagStim, from 1 to 9.

4 The second serial line port for dual 200^2 mode, from 1 to 9.

For a CED 3304 only:

1 CED 3304 in use. -1=Do not use, 0=Use CED 3304.

2 The device flags. This is 1 for the high level trigger, 0 for low trigger.

3 The serial line port (COM port) used to control the 3304, from 1 to 9.

4 The range switch setting, from 0 (10 microamps) to 3 (10 milliamps).

`val` If present, this sets the new value of the selected parameter, otherwise it will be unchanged. It is not possible to set a new value for parameters zero, this is set by the type of auxiliary device support installed.

Returns The parameter value selected at the time of the function call.

See also: `SampleAuxStateValue(), SampleStates(), SampleState()`

SampleAuxStateValue()

This function is used to get or set auxiliary states device settings for individual states. The meaning and use of these settings varies according to the type of auxiliary states device in use, it is vitally important that you use `SampleAuxStateParam(0)` in your script to check that the correct auxiliary states device is installed. At the time of writing the auxiliary devices supported are the MagStim and the CED 3304.

```
Func SampleAuxStateValue(state%, num%, {val});
```

`state%` The state number for which the settings are being read or written, from zero to the number of extra states enabled.

`num%` This selects the setting that will be read or set. The meaning and use of all settings varies with the type of auxiliary states device in use.

For a MagStim only:

0 The state flags. This is 1 if manual control is selected, otherwise zero.

1 The power level from zero to 100 percent (or 110 percent for Rapid^2 single pulse mode).

2 The secondary power level for BiStim^2 and dual 200^2 devices.

3 The pulse interval for BiStim^2 devices.

4 The pulse frequency in Hz for Rapid^2 devices.

5 The number of pulses for Rapid^2 devices.

For a CED 3304 only:

0 The state flags. Unused with the 3304 and read as 0.

1 The current in uA from zero to the maximum allowed by the range setting.

`val` If present, this sets the new value of the selected setting, otherwise it will be unchanged.

Returns The setting value selected at the time of the function call.

See also: `SampleAuxStateParam(), SampleStates(), SampleState()`

SampleBar()

This gives you access to the Sample toolbar. The format of strings passed by this command is the button label (up to 8 characters), followed by a vertical bar, followed by the full path name to a sampling configuration file, including the `.sgc` file extension, followed by a vertical bar, then a comment to display when the mouse pointer is over the button. If you call the command with no arguments it returns the number of buttons in the toolbar.

```
Func SampleBar({n% {, &get$}});
Func SampleBar(set$);
```

`n%` If set to -1, `get$` must be omitted, all buttons are cleared and the function returns 0. When set to the number of a button (the first button is 0), `get$` is as described above. In this case, the function returns -1 if the button does not exist, 0 if it exists and is the last button, and 1 if higher-numbered buttons exist.

`set$` The string passed in should have the format described above. The function returns the new number of buttons or -1 if all buttons are already used.

Returns See the descriptions above. Negative return values indicate an error.

For example, the following code clears the script bar and sets two buttons:

```
SampleBar(-1);     'clear all buttons
SampleBar("Fast|C:\\Signal3\\Fast.sgc|Fast 4 channel sampling");
SampleBar("Faster|C:\\Signal3\\FastXX.sgc|Very fast sampling");
```

See also: `App()`

SampleBurst()

This gets or sets burst mode sampling as seen in the sampling configuration dialog.

```
Func SampleBurst({bMode%});
```

`bMode%` If present and non-zero this turns burst mode on in the sampling configuration. Burst mode is often to be preferred, as the actual sampling rate used is more likely to match the preferred rate set.

Returns 1 if burst mode is on, 0 if it is off.

See also: `SampleClear()`, `SampleRate()`, `SamplePoints()`, `SampleTrigger()`, `SamplePorts()`

SampleClear()

This procedure sets the contents shown in all the panes of the sampling configuration dialog to a standard state. You can use the sampling configuration commands to get or change values in the sampling configuration. There is a full list of the sampling configuration commands in the *Commands by function* chapter.

```
Proc SampleClear();
```

SampleDacFull()

This function gets the full-scale value used to scale values written to the DACs from the sampling configuration and optionally sets it to a new value.

```
Func SampleDacFull(port {, new});
```

`port` The DAC number, from 0 to 7.

`new` If present, sets the value in the units for this DAC corresponding to a full-scale value. This value is used throughout Signal to calibrate DAC values.

Returns The DAC full-scale value before the call.

See also: `Pulses()`, `SampleDacMask()`, `SampleDacZero()`, `SampleDacUnits$()`, `SampleStateDac()`

SampleDacMask()

This function gets the mask value used to enable the DAC outputs from the sampling configuration and optionally sets it to a new value.

```
Func SampleDacMask({new});
```

`new` If present, sets the mask enabling the DAC outputs. This mask has one bit for each DAC, set bits enable the corresponding DAC output.

Returns The DAC outputs mask value before the call.

See also: `SampleStatesMode()`, `SampleDigOMask()`, `SampleDigIMask()`, `SampleStateDac()`

SampleDacUnits$()

This function gets the units string for a DAC from the sampling configuration and optionally sets it to a new value.

```
Func SampleDacUnits$(port {, new$});
```

`port` The DAC number, from 0 to 7.

`new` If present, sets the units string for this DAC. This value is used throughout Signal to calibrate DAC values.

Returns The DAC units string before the call.

See also: `Pulses(), SampleDacMask(), SampleDacFull(), SampleDacZero(), SampleStateDac()`

SampleDacZero()

This function gets the zero value used to scale values written to the DACs from the sampling configuration and optionally sets it to a new value.

```
Func SampleDacZero(port {, new});
```

port The DAC number, from 0 to 7.

`new` If present, sets the value in the units for this DAC corresponding to a zero value. This value is used throughout Signal to calibrate DAC values.

Returns The DAC zero value before the call.

See also: `Pulses(), SampleDacMask(), SampleDacFull(), SampleDacUnits$(), SampleStateDac()`

SampleDigIMask()

This function gets the mask value used to enable the digital inputs for the External digital multiple states mode from the sampling configuration and optionally sets it to a new value.

```
Func SampleDigIMask({new});
```

`new` If present, sets the mask enabling the digital inputs. This mask has one bit for each digital input, set bits enabling the corresponding inputs. This value is only used in External digital states mode.

Returns The digital inputs mask value before the call.

See also: `SampleStatesMode(), SampleStateDig(), SampleDacMask(), SampleDigOMask()`

SampleDigMark()

This function sets and gets the flag for enabling the digital marker channel in the sampling configuration.

```
Func SampleDigMark({on%});
```

`on%` If present and non-zero, enables the marker channel

Returns 1 if the marker channel was on, 0 if it was off.

See also: `SampleClear(), SampleKeyMark()`

SampleDigOMask()

This function gets the mask value used to enable the digital outputs from the sampling configuration and optionally sets it to a new value.

```
Func SampleDigOMask({new});
```

`new` If present, sets the mask enabling the digital outputs. This mask has one bit for each digital output, set bits enabling the corresponding outputs.

Returns The digital outputs mask value before the call.

See also: `SampleStatesMode(), SampleStateDig(), SampleDacMask(), SampleDigIMask()`

SampleFixedInt()

This function sets and gets the sweep interval for Fixed interval and Fast Fixed int sweep modes, as stored in the pulses information. Note that this sets the interval after frames with the specified state, not the interval before a frame. This function normally operates on the stored sampling configuration but, if used during sampling, it operates upon the on-going sampling.

```
Func SampleFixedInt(state {, period});
```

`state` This sets the state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`period` If present, this argument sets the fixed interval period, in seconds.

Returns The fixed interval period at the time of the call.

See also: `SampleOutMode(), SampleOutTrig(), SampleOutClock(), SampleFixedVar(), Pulses(), SampleStates()`

SampleFixedVar()

This function sets and gets the percentage variation of the sweep interval for Fixed interval sweep mode, as stored in the pulses information. This function normally operates on the stored sampling configuration but, if used during sampling, it operates upon the on-going sampling.

```
Func SampleFixedVar(state {, vary});
```

`state` This sets the state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`vary` If present, this argument sets the fixed interval variation, from 0 to 100 percent.

Returns The fixed interval variation percentage at the time of the call.

See also: `SampleOutMode(), SampleOutTrig(), SampleOutClock(), SampleFixedInt(), Pulses(), SampleStates()`

SampleHandle()

This gets the handle of a view associated with sampling. This can be used to position, show or hide the sampling control panel or the output control panel.

```
Func SampleHandle(which%);
```

`which%` Selects which view handle to return:

0 Main file view.
1 Sampling control panel.
2 Sequencer control panel.
3 Pulses configuration dialog.
4 States control bar.

Returns The view handle or 0 if the view does not exist.

See also: `View(), ViewList(), Window(), WindowVisible()`

SampleKey()

This procedure adds events to the keyboard marker channel, exactly as if you had typed them (with the sampling document view as the current view). If there is no sampling, the procedure does nothing. If the output sequencer is running, and you add a key that corresponds to a key linked to a sequencer step, the sequencer jumps to the step.

```
Func SampleKey(key$);
```

`key$` The first character of the string is added to the keyboard marker channel.

Returns The time in seconds at which the marker was added to the keyboard marker channel.

See also: `SampleClear(), SampleKeyMark()`

SampleKeyMark()

This function turns the keyboard marker channel on or off or gets the current setting of this from the sampling configuration.

```
Func SampleKeyMark(on%);
```

`on%` If present and non-zero this turns keyboard markers on in the sampling configuration.

Returns 1 if the keyboard channel is on in the sampling configuration, 0 if not.

See also: `SampleClear(), SampleDigMark(), SampleKey()`

SampleLimitFrames()

This function corresponds to the Number of Frames field on the Automation page of the sampling configuration dialog.

```
Func SampleLimitFrames({limit%});
```

`limit%` The number of frames to set as a limit. A positive number sets the limit and checks the box (enabling the limit). A negative number sets the limit to the positive time, but clears the check box (so the limit is not used). A value of zero or omitting the argument, leaves the time limit unchanged.

Returns The function returns the frames limit as it was at the time of the call. If the limit is disabled, the number of frames is returned negated.

See also: `SampleClear(), SampleLimitSize(),SampleLimitTime(), SampleWrite()`

SampleLimitSize()

This function corresponds to the File size field on the Automation page of the sampling configuration dialog.

```
Func SampleLimitSize({size});
```

`size` The size limit for the output file, in KB. A positive value sets the size and enables the limit. A negative value sets the limit to the positive value of size, but disables the limit. A zero value, or omitting the argument, means no change.

Returns The limit before the call. If the limit is disabled, the value is returned negated.

See also: `SampleClear(), SampleLimitFrames(), SampleLimitTime(), SampleWrite()`

SampleLimitTime()

This function corresponds to the Run time field on the Automation page of the sampling configuration dialog.

```
Func SampleLimitTime({time});
```

`time` The time in seconds to set as a limit. A positive `time` sets the limit and checks the box (enabling the limit). A negative `time` sets the limit to the positive time, but clears the check box (so the limit is not used). A value of zero, or omitting the argument, leaves the time limit unchanged.

Returns The function returns the time limit as it was in the sampling configuration at the time of the call. If the limit is disabled, the time is returned negated.

See also: `SampleClear()`, `SampleLimitFrames()`, `SampleLimitSize()`, `SampleWrite()`

SampleMode()

This function sets and gets the sampling sweep mode as seen in the sampling configuration dialog.

```
Func SampleMode({mode%});
```

`mode%` This argument determines the action of the command:
0 Sets basic post triggered mode.
1 Sets peri triggered mode.
2 Sets outputs frame mode.
3 Sets fixed interval mode.
4 Fast triggers mode.
5 Fast fixed interval mode.
6 For future expansion.

Returns The sweep mode from the sampling configuration at the time of the call.

See also: `SampleClear()`, `SamplePeriType()`, `SampleFixedInt()`, `SampleOutLength()`, `SampleOutTrig()`

SampleOutClock()

This function sets and gets the outputs clock as seen in the sampling configuration dialog Outputs page.

```
Func SampleOutClock({period {, synch%}});
```

`period` If present, this argument sets the outputs clock period, in seconds. This value sets the time resolution for pulses and sequencer output, for measuring sweep absolute start times, for timing sweeps in Fixed interval mode and for measuring the time of marker data. For the standard 1401, this value should not be less than 10 ms, for a 1401*plus* 3 ms, for a micro1401 0.1 ms, for a Micro1401 mk II 25 µs and for a Power1401 10 µs.

`synch%` If present, this sets whether to synchronise pulse outputs to the sampling sweep. A non-zero value is equivalent to checking the "`Synchronise sampling`" box in the sampling configuration dialog.

Returns The outputs clock period from the configuration at the time of the call.

See also: `SampleOutMode()`, `SampleFixedInt()`, `SampleOutTrig()`, `Pulses()`

SampleOutLength()

This function sets and gets the length of the pulses output frame; the length of time that pulses are generated, as stored in the pulses information for use in Outputs frame and Fixed interval sweep modes. This function normally operates on the stored sampling configuration but, if used during sampling, it operates upon the on-going sampling.

```
Func SampleOutLength(state {, length});
```

`state` This sets the state (pulses set) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`length` If present, this argument sets the length of the pulses output frame, in seconds.

Returns The pulses output frame length at the time of the call.

See also: `SampleOutMode(), SampleOutTrig(), SampleOutClock(), Pulses(), SampleStates()`

SampleOutMode()

This function sets and gets the outputs mode as seen in the sampling configuration dialog Outputs page.

```
Func SampleOutMode({mode%});
```

`mode%` This argument determines the action of the command:

0 Sets None outputs mode.
1 Sets Pulses outputs mode.
2 Sets Sequencer outputs mode.
3 For future expansion.

Returns The outputs mode from the configuration at the time of the call.

See also: `SampleClear(), Pulses()`

SampleOutTrig()

This function sets and gets the sweep trigger time within the pulses frame as stored in the pulses information for use in Outputs frame and Fixed interval sweep modes. This function normally operates on the stored sampling configuration but, if used during sampling, it operates upon the on-going sampling.

```
Func SampleOutTrig(state {, time});
```

`state` This sets the state (set of pulses) to which this applies, from 0 to 256. Use 0 if multiple states are not in use.

`time` If present, this argument sets the sweep trigger time within the pulses output frame, in seconds. The value should be from 0 to the pulses output frame length.

Returns The sweep trigger time at the time of the call.

See also: `SampleOutLength(), SampleFixedInt(), SampleOutClock(), Pulses(), SampleStates()`

SamplePause()

This function ascertains or sets whether the sampling mode is paused, which is the state of the Pause at sweep end check box.

```
Func SamplePause({pause%});
```

`pause%` If present this is equivalent to changing the state of the Pause at sweep end check box. A non-zero value pauses sampling and zero enables sampling.

Returns If `pause%` is present it returns the new state as 1 or 0, or a negative error code. If `pause%` is absent it returns the current state as 1 or 0.

See also: `SampleSweep(), SampleStart(), SampleStop(), SampleStatus(), SampleWrite(), SampleAbort()`

SamplePeriBitState()

This function gets or selects the state of the digital input bit required to trigger sampling. It is equivalent to the dropdown selection box in the digital peri-trigger sampling configuration.

```
Func SamplePeriBitState({set%});
```

`set%` If present a value of 1 selects Trigger on bit high in the digital peri-trigger sampling configuration. A value of 0 selects Trigger on bit low.

Returns The value for the setting at the time of the call.

See also: `SamplePeriDigBit(), SamplePeriHyst(), SamplePeriLevel(), SamplePeriLowLev(), SamplePeriType(), SamplePeriPoints()`

SamplePeriDigBit()

This function gets or sets the digital bit number in the digital peri-trigger information within the sampling configuration.

```
Func SamplePeriDigBit({bit%});
```

`bit%` If present and in the range 8-15, this sets the new digital bit number in the digital peri-trigger sampling configuration.

Returns The value for the peri-trigger digital bit in the sampling configuration at the time of the call.

See also: `SamplePeriBitState(), SamplePeriHyst(), SamplePeriLevel(), SamplePeriLowLev(), SamplePeriType(), SamplePeriPoints()`

SamplePeriHyst()

This function gets or sets the hysteresis value for triggering from an analogue channel level in the peri-trigger information within the sampling configuration. If used while sampling is in progress, this gets and sets the hysteresis value currently in use.

```
Func SamplePeriHyst({level});
```

`level` If present this sets the new hysteresis value in the peri-trigger sampling configuration or sampling document.

Returns The value for hysteresis in the sampling configuration at the time of the call.

See also: `SamplePeriDigBit(), SamplePeriBitState(), SamplePeriLevel(), SamplePeriLowLev(), SamplePeriType(), SamplePeriPoints()`

SamplePeriLevel()

This function gets or sets the threshold level for triggering from an analogue channel level in peri-trigger mode. This is the threshold level for +Analogue and -Analogue peri-trigger types and the upper threshold for the =Analogue peri-trigger type. If used while sampling is in progress, this gets and sets the threshold level currently in use.

```
Func SamplePeriLevel({level});
```

`level` If present this sets the peri-trigger threshold level in the sampling configuration or sampling document.

Returns The value for peri-trigger threshold level in the sampling configuration.

See also: `SamplePeriDigBit(), SamplePeriBitState(), SamplePeriHyst(), SamplePeriLowLev(), SamplePeriType(), SamplePeriPoints()`

SamplePeriLowLev()

This function gets or sets the peri-trigger Lower threshold for =Analogue peri-trigger type in the peri-trigger sampling configuration. If used while sampling is in progress, this gets and sets the lower threshold level currently in use.

```
Func SamplePeriLowLev({level});
```

`level` If present this sets the lower threshold for =Analogue peri-trigger type in the peri-trigger sampling configuration or sampling document.

Returns The value for the lower threshold for =Analogue peri-trigger type in the peri-trigger sampling configuration.

See also: `SamplePeriDigBit(), SamplePeriBitState(), SamplePeriHyst(), SamplePeriLevel(), SamplePeriType(), SamplePeriPoints()`

SamplePeriType()

This function gets or sets the type of peri-trigger in the peri-trigger information within the sampling configuration.

```
Func SamplePeriType({pts%});
```

`pts%` If present this sets the type of trigger in the sampling configuration as follows:

0 +Analogue.
1 -Analogue.
2 =Analogue.
3 Digital.
4 Event.

Returns The trigger type in the sampling configuration at the time of the call.

See also: `SamplePeriDigBit(), SamplePeriBitState(), SamplePeriHyst(), SamplePeriLevel(), SamplePeriLowLev(), SamplePeriPoints()`

SamplePeriPoints()

This function gets or sets the number of data points in the frame before the trigger as given by Pre-trig. points in the Peri-trigger section of the sampling configuration.

```
Func SamplePeriPoints({pts%});
```

`pts%` If present this sets the pre-trigger points in the peri-trigger sampling configuration. This can be any negative number or a positive number less than the points per sweep.

Returns The value for pre-trigger points in the peri-trigger sampling configuration.

See also: `SamplePeriDigBit(), SamplePeriBitState(), SamplePeriHyst(), SamplePeriLevel(), SamplePeriLowLev(), SamplePeriType()`

SamplePoints()

This function gets or sets the number of data points per ADC port per frame as given by the Frame points in the sampling configuration.

```
Func SamplePoints({pts%});
```

`pts%` If present this sets the number of frame points in the sampling configuration.

Returns The value for frame points in the sampling configuration.

See also: `SampleClear(), SampleRate(), SamplePeriPoints(), SampleTrigger()`

SamplePortFull()

This function gets and sets the Full value for an input port, as shown in the sampling configuration dialog. Complete calibration of a waveform channel requires Full, Zero and Units to be set up correctly. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling.

```
Func SamplePortFull(port% {,full});
```

`port%` The port number (0-79).

`full` The value of the data corresponding to the ADC full-scale level (+5 volts or +10 volts for a 10 volt system) at the input.

Returns The value for the port Full scale value at the time of the call, or zero for illegal port numbers.

See also: `SampleClear(), SamplePorts(), SamplePortOptions$(), SamplePortUnits$(), SamplePortZero(), SampleTel()`

SamplePortName$()

This function gets and sets the title attached to a port, as shown in the sampling configuration dialog.

```
Func SamplePortName$(port% {,new$});
```

`port%` The port number (0-79).

`new$` If present, the new title. If the title is too long, it is truncated.

Returns The title at the time of the call, or an empty string for illegal channel numbers.

See also: `SampleClear(), SamplePorts(), SamplePortFull(), SamplePortOptions$(), SamplePortUnits$(), SamplePortZero()`

SamplePortOptions$()

This function gets and sets the online processing options attached to a waveform input port setup, as shown in the sampling configuration dialog.

```
Func SamplePortOptions$(port% {,new$});
```

`port%` The port number (0-79).

`new$` If present, the new options string, up to 7 characters long. If the string is too long it will be truncated.

Returns The options string at the time of the call, or an empty string for illegal channel numbers.

See also: `SampleClear(), SamplePortFull(), SamplePorts(), SamplePortName$(), SamplePortUnits$(), SamplePortZero()`

SamplePorts()

This function gets or sets the ADC ports to be used as shown in the sampling configuration.

```
Func SamplePorts({get%[]|num%{,new%[]}});
```

`num%` If present this sets the number of ADC ports to take from the `new%` array. If `new%` is not present the new ADC ports will be `0..num%-1`.

`get%[]` If present as a single argument this is filled with ADC ports from the sampling configuration up to the size of the array. If there are insufficient ports to fill the array the unused entries are left unchanged.

`new%[]` If present this holds the new ADC ports for the sampling configuration. The number of new ADC ports will be restricted by the size of `num%` or by the size of this array, whichever is the smaller.

Returns The number of ADC ports at the time of the call.

See also: `SampleClear(), SamplePortFull(), SamplePortName$(), SamplePortOptions$(), SamplePortUnits$(), SamplePortZero()`

SamplePortUnits$()

This function gets and sets the units for an input port, as shown in the sampling configuration dialog. Complete calibration of a waveform channel requires Full, Zero and Units to be set up correctly.

```
Func SamplePortUnits$(port% {,new$});
```

`port%` The port number (0-79).

`new$` The units to use. If the string is longer than 7 characters only the first 7 are used.

Returns The units at the time of the call, or an empty string for illegal channel numbers.

See also: `SampleClear(), SamplePorts(), SamplePortFull(), SamplePortOptions$(), SamplePortZero()`

SamplePortZero()

This function gets and sets the Zero value for an input port, as shown in the sampling configuration dialog. This function normally operates on the stored sampling configuration but if used during sampling it operates upon the on-going sampling.

```
Func SamplePortZero(port% {,zero});
```

`port%` The port number (0-79).

`zero` The value of the data corresponding to a zero-volt input at the ADC.

Returns The value for the port zero value at the time of the call, or zero for illegal port numbers.

See also: `SampleClear(), SamplePorts(), SamplePortFull(), SamplePortUnits$(), SamplePortOptions$()`

SampleProtocol()

This function is used during sampling to start off execution of a protocol to from the protocols defined in the sampling configuration.

```
Func SampleProtocol(num|name$);
```

num The protocol number to use, from 1 to the number returned by `Protocols()`.

name$ The name of the protocol to use.

Returns The number of the protocol in use before this call or a negative error code.

See also: `Protocols(), ProtocolName$(), SampleState()`

SampleRate()

This function gets the sample rate in Hz from the sampling configuration and optionally sets it to a new preferred value.

```
Func SampleRate({new});
```

`new` If present, the new preferred rate in Hz. The actual sampling rate used will be as close as possible to new rate, but it will not always match exactly.

Returns The sampling rate before the call. This is the actual sampling rate that would have been used, not the preferred rate.

See also: `SampleClear(), SamplePoints(), SampleTrigger(), SampleBurst(), BinSize()`

SampleReset()

This function can be used while sampling is in progress to abandon sampling, delete any data that has been written to disk, and return to the state as if `FileNew()` had just been used to create a new data file.

```
Func SampleReset();
```

Returns 0 if the reset operation completed without a problem, or a negative error code.

See also: `SampleAbort(), SampleStart(), SampleStop(), SampleStatus(), SampleWrite(), SamplePause()`

SampleSeqCtrl()

This sets and gets options that control the use of the output sequence. Currently there is only one option.

```
Func SampleSeqCtrl(opt%{, new%});
```

`opt%` There is currently only one option: 1 = get or set the sequencer jump control. This is the same as the Sequencer jumps controlled by setting in the Sequencer tab of the Sampling configuration dialog.

`new%` The new value for the control option. For the jump control: 0 = keyboard, control panel and script, 1 = control panel and script, 2 = script only.

Returns If you are setting a value or if this is used at an inappropriate time, the function returns 0. If you are reading a value, the function returns the value.

See also: `SampleKey()`, `SampleSeqStep()`, `SampleSequencer$()`, `SampleStart()`

SampleSeqStep()

This returns the current sequencer step or -1 if not sampling. If no sequence is running the result is usually 0 (but this is not guaranteed).

```
Func SampleSeqStep();
```

Returns The current sequence step number, or -1 if not sampling.

See also: `SampleKey()`, `SampleSequencer$()`, `SampleSeqVar()`

SampleSeqTable()

If there is a sampling document with an output sequence, you can use this function to find the size of any table set in the sequence by the `TABSZ` directive. You can also use this to transfer data between an integer array and the table.

```
Func SampleSeqTable({tab%[]{, offs%{, get%}}});
```

`tab%[]` An integer array holding items to transfer to the 1401 sequencer table or to hold items read back from the table. The array size sets the maximum item count.

`offs%` This sets the index into the sequencer table to start the transfer. The first index in the table is 0. If this value is negative or greater than or equal to the sequencer table size, no data is transferred. If omitted, the value 0 is used.

`get%` Set 0 or omit this argument to transfer data to the sequencer table, set to 1 to transfer data from the sequencer table.

Returns If you call this with no arguments, the return value is the size of the sequencer table. Otherwise, the returned value is the number of items transferred between the sequencer table and the array. A negative error code is also possible, for example -1 if there is no sampling document.

See also: `SampleKey()`, `SampleSequencer$()`, `SampleSeqVar()`

SampleSequencer()

You can use this function to set the sequencer file to attach to the Sampling configuration. Use `SampleSequencer$()` to get the name of the current sequencer file.

```
Func SampleSequencer(new$);
```

`new$` The name of the sequence file. Pass an empty string to set no sequencer file.

Returns It returns 0 if all was well, or a negative error code.

See also: `SampleKey()`, `SampleSequencer$()`, `SampleSeqVar()`

SampleSequencer$()

This function returns the name of the sequencer file that is currently attached to the sampling configuration. Use `SampleSequencer()` to set the file.

```
Func SampleSequencer$();
```

Returns It returns the current sequencer file name, or an empty string if there is no file. The returned name includes the full path.

See also: `SampleKey(), SampleSequencer(), SampleSeqVar()`

SampleSeqVar()

This is used during sampling with an output sequence, to get or set the value of an output sequencer variable. Values set before the sampling window exists are ignored. Values set before `SampleStart()` set the initial variable values.

```
Func SampleSeqVar(sVar%{, new%});
```

`sVar%` The sequencer variable to set or read, in the range 1 to 64.

`new%` The new value for the output sequencer variable. If present, the value of the variable is updated. Omit to return the variable value. A common error when setting variables for the `DAC` instruction is to set a value 65536 times too small.

Returns If you are setting a value or if this is used at an inappropriate time, the function returns 0. If you are reading a value, the function returns the value.

See also: `SampleKey(), SampleSeqStep(), SampleSequencer$(), SampleStart(), SampleSeqWave()`

SampleSeqWave()

This command can be used to get or set the size of the arbitrary waveform output memory area used by the Signal outputs sequencer, or to load waveform data into or out of the waveform area. If a single DAC is being used for waveform output, the array data is a simple list of values. For more than one DAC, the values for the various DACs are interleaved.

```
Func SampleSeqWave(area%{, arr%[]{, offs%{, get%});
Func SampleSeqWave(area%{, arr[]{, offs%{, get%});
```

These two forms of the command copy data between a script array and the waveform output area. They can be used while sampling is in progress, or after the `FileNew()` function has been used but before `SampleStart()`, but will not have any effect if used when sampling has not been set up.

`area%` The waveform area number, must be set to zero.

`arr%[]` The array holding, or to be updated with, the waveform data. For integer arrays values from -32768 to 32767 span the complete DAC range.

`arr[]` As for `arr%[]` but an array of real numbers. For real arrays, the DAC scaling values for the sequencer will be used to convert from user units in the array to DAC values.

`offs%` The offset within the waveform area to start the transfer. The size of the data array sets the transfer size. If this parameter is omitted, a value of zero is used.

`get%` Set this parameter to 1 to read data from the waveform area, set it to zero or omit it to transfer data into the waveform area.

```
Func SampleSeqWave(area%, mask%{, pnts%, hz%});
Func SampleSeqWave(area%, num%);
```

This form of the command is either used to define the waveform area, or to retrieve the area settings. The form with four arguments defines the waveform area:

`mask%` A value that defines which DACs are used, with bit zero set if DAC 0 is used, bit 1 for DAC 1, bit 2 for DAC 2 and bit 3 for DAC 3.

`pnts%` Sets the number of waveform points per DAC.

`hz%` Sets the waveform output rate, in points per second.

`num%` A number which determines the return value of the function. Possible values for `num%` are:

0 `points%` * number of DACs, i.e. the total data size.
-1 `mask%`.
-2 `pnts%`.
-3 `hz%`.
-4 The number of DACs in `mask%`.

Returns If only `area%` is provided then the return code is the area size. If `num%` is provided the return value is as above. In all other cases the return value is 0 if all went well or a negative error code.

See also: `SampleKey()`, `SampleSeqStep()`, `SampleSequencer$()`, `SampleStart()`, `SampleSeqVar()`

SampleStart()

This function can be used after `FileNew()` has created a new file view based on the current sampling configuration. It starts sampling immediately or on a 1401 event trigger.

```
Func SampleStart({trig%});
```

`trig%` If this is 0 or omitted, sampling starts immediately, otherwise sampling waits for a trigger signal on the 1401 E1 input.

Returns 0 if all went well or a negative error code.

See also: `SampleAbort()`, `SampleReset()`, `SampleStop()`, `SampleStatus()`, `SampleWrite()`, `SamplePause()`

SampleState()

This function is used during sampling to set the current state directly; it is the scripting equivalent of controlling the state manually with the states control bar. The command should be used with states sequencing set to Manual, or the values set will be overridden by the sampling system.

```
Func SampleState(num);
```

num The state to use, from 0 to 256.

Returns The state in use before this call or a negative error code.

See also: `SampleStatesOrder()`, `SampleStatesRun()`, `SampleStates()`

SampleStateDac()

This function gets the DAC output value for a specific state from the sampling configuration and optionally sets it to a new value.

```
Func SampleStateDac(state, port {, new});
```

`state` The state for which information is required, from 0 to 256.

`port` The DAC number, from 0 to 3.

`new` If present, sets the value to be output to the DAC for this state in Static outputs states mode.

Returns The DAC output value before the call.

See also: `SampleStatesMode(), SampleStates(), SampleState(), SampleDacMask(), SampleDacFull(), SampleDacZero()`

SampleStateDig()

This function gets the value to be written to the digital outputs for a specific state or the value read from the digital inputs to set a state and optionally sets it to a new value.

```
Func SampleStateDig(state {, new});
```

`state` The state for which information is required, from 0 to 256.

`new` If present, sets the value to be written to the digital outputs for this state in Static outputs states mode, or the value used to test the digital input data in External digital states mode.

Returns The digital output or input value before the call.

See also: `SampleStatesMode(), SampleStates(), SampleState(), SampleStateDac(), SampleDigIMask()`

SampleStateLabel$()

This function gets the optional label for a specific state (used to label the state control bar buttons, among other things) and optionally sets it to a new value.

```
Func SampleStateLabel$(state {, new$});
```

`state` The state for which information is required, from 0 to 256.

`new$` If present, sets the new label for this state. A state label should consist of printable text and should not be longer than ten characters.

Returns The label for the state before the call.

See also: `SampleStateRepeats(), SampleStates(), SampleState(), SampleStateDac()`

SampleStateRepeats()

This function gets the number of repeats for a specific state from the sampling configuration and optionally sets it to a new value.

```
Func SampleStateRepeats(state {, new});
```

`state` The state for which information is required, from 0 to 256.

`new` If present, sets the number of times the state is repeated in Numeric and Random ordering mode if Individual repeats is enabled.

Returns The number of repeats before the call.

See also: `SampleStateRepeats(), SampleStates(), SampleState()`

SampleStates()

This function gets the number of extra states from the sampling configuration and optionally sets it to a new value.

```
Func SampleStates({new});
```

`new` If present, the new number of extra states. Values from 1 to 256 set the states and turn on Multiple states mode, a value of zero turns off multiple states.

Returns The number of extra states before the call, or zero if multiple states were disabled.

See also: `SampleStatesMode()`, `SampleStatesOrder()`, `SampleState()`

SampleStatesIdle()

This function gets the count of states ordering cycles before idling from the sampling configuration and optionally sets it to a new value.

```
Func SampleStatesIdle({idle%});
```

`idle%` If present, this sets the number of cycles of states using Numeric or Random state sequencing before idling. A zero value means keep cycling forever. This parameter is ignored for protocol ordering.

Returns The states cycles before idling before the call.

See also: `SampleStates()`, `SampleStatesMode()`, `SampleStatesOrder()`, `SampleState()`, `SampleProtocol()`, `ProtocolFlags()`

SampleStatesMode()

This function gets the mode for multiple states from the sampling configuration and optionally sets it to a new value.

```
Func SampleStatesMode({new});
```

`new` If present, sets the new multiple states mode as follows:
0 External digital
1 Static outputs
2 Dynamic outputs

Returns The mode for multiple states before the call.

See also: `SampleStates()`, `SampleStatesOrder()`, `SampleState()`

SampleStatesOrder()

This function gets the ordering mode for multiple states from the sampling configuration and optionally sets it to a new value.

```
Func SampleStatesOrder({new%});
```

`new%` If present, this sets the new multiple states ordering mode as follows:
0 Numeric 1 Random 2 Protocol 3 Semi-random

Returns The states ordering mode before the call.

See also: `SampleStates()`, `SampleStatesIdle()`, `SampleStatesMode()`, `SampleState()`, `SampleProtocol()`, `ProtocolFlags()`

SampleStatesPause()

This function gets the paused or not-paused state of multiple states cycling and optionally sets it to a new value.

```
Func SampleStatesPause({pause%});
```

`pause%` If present, this sets the new paused condition of state cycling:
0 Not paused
1 Paused

Returns The paused condition before the call.

See also: `SampleStates(), SampleStatesIdle(), SampleStatesMode(), SampleState(), SampleProtocol(), ProtocolFlags()`

SampleStatesRepeats()

This function gets the number of times each state is repeated from the sampling configuration and optionally sets it to a new value, or selects individual repeats mode.

```
Func SampleStatesRepeats({new%});
```

`new%` If present and non-zero, disables individual repeats and sets the number of times each state is repeated in Numeric and Random ordering mode. A value of zero turns on individual repeats.

Returns The number of repeats before the call, or zero if individual repeats were enabled.

See also: `SampleStateRepeats(), SampleStatesOrder(), SampleState()`

SampleStatesReset()

This function is used during sampling to reset the states-sequencing system and the pulses built-in varations; it is the scripting equivalent of pressing the Reset button on the states control bar.

```
Func SampleStatesReset();
```

Returns Zero or a negative error code.

See also: `SampleStatesOrder(), SampleStatesRun(), SampleStates()`

SampleStatesRun()

This function is used during sampling to set the state seqencing execution mode; the scripting equivalent of using the Manual, On write or Cycle buttons on the states control bar.

```
Func SampleStatesRun({mode%});
```

`mode%` The mode of sequencing to use, as follows:
0 Manual or direct script control of states
1 States sequencer runs, moves to next state if data written
2 States sequencer runs, moves to next state unconditionally

Returns The state sequencing mode in use before this call or a negative error code.

When using protocols, setting mode to 0 will terminate the protocol though this command cannot be used to start a protocol ruinning.

See also: `SampleStatesOrder(), SampleStatesReset(), SampleState()`

SampleStatus()

This function enquires about the state of any sampling.

```
Func SampleStatus();
```

Returns A code indicating the sampling state or -1 if there is no sampling:

0 A file view is ready to sample, but it has not been told to start yet.
1 Sampling is waiting for an Event 1 trigger.
2 Sampling of a sweep is now in progress or is awaiting a trigger.
3 Sampling is paused at the end of a sweep.
4 Sampling is stopped but not finished (changes to -1 when it has finished).

See also: `SampleAbort(), SampleReset(), SampleStart(), SampleStop(), SampleWrite(), SamplePause()`

SampleStop()

This function stops sampling in progress and is equivalent to using the Stop and Finish buttons of the sampling control panel. The default behaviour is that there is no intermediate state between stopping and finishing, when sampling is stopped by using this function. The function does not return until sampling has stopped.

```
Func SampleStop({noFin%});
```

`noFin%` If present and non zero then sampling will stop but not finish. `SampleSweep()` may then be used to continue sampling.

Returns 0 if sampling stopped correctly or a negative error code.

See also: `SampleAbort(),SamplePause(), SampleReset(), SampleStart(),SampleStatus(), SampleSweep(), SampleWrite()`

SampleSweep()

If sampling is paused at the end of a sweep, or stopped but not finished because a limit was reached, this starts sampling of the next sweep. The current sweep will be lost if it is unsaved. This function is the equivalent of the Continue button in the sampling control panel (or More when sampling is stopped).

```
Func SampleSweep();
```

Returns 0 or a negative error code.

See also: `SamplePause(), SampleReset(), SampleStart(), SampleStop(), SampleWrite()`

SampleTel()

This function gets or sets the telegraph options in the sampling configuration.

```
Func SampleTel(nSet%, nPort%, nTel%{, volt[]|volt, gain[]|gain});
Func SampleTel(nSet%, opt%);
```

`nSet%` The set of telegraph values to work with.

`nPort%` The port number whose input is to be scaled by the telegraph. If this and `nTel%` are both set to -1 then these telegraph settings will be turned off.

`opt%` This can be set to the following negative values which cause the function to return the following values:

- -1 The scaled port number for these settings.
- -2 The telegraph port number.
- -3 The number of items in the telegraph list.
- -4 The first telegraph voltage in the list.
- -5 The first gain in the list.
- -n If n > 3 and n is even, the (n-2)/2nd voltage in the list is returned. If n > 4 and n is odd, the (n-3)/2nd gain in the list is returned.

`nTel%` The port number being used to read the telegraph voltage. If this and `nPort%` are both set to -1 then these telegraph settings will be turned off.

`volt` A telegraph voltage. If this is set to 0 and no gain is given then the list of gains and telegraph voltages is cleared.

`volt[]` An array of telegraph voltages.

`gain` The gain associated with a particular voltage.

`gain[]` An array of gains associated with the array of telegraph voltages provided.

Returns 0, the requested value or a negative error code.

See also: `SampleClear()`, `SamplePorts()`, `SamplePortFull()`, `SamplePortZero()`

SampleTrigger()

This function gets or sets the external trigger option in the sampling configuration.

```
Func SampleTrigger({trig%});
```

`trig%` If this is non-zero, sampling of each frame waits for a trigger input. Zero turns trigger mode off.

Returns 0 or 1, or a negative error code.

See also: `SampleClear()`, `SampleRate()`, `SampleSweep()`, `SamplePoints()`, `SampleStatus()`

SampleWrite()

This function controls the automatic writing of data to the file during sampling and is equivalent to the Write to disk at sweep end check boxes.

```
Func SampleWrite({write%});
```

`write%` If present this sets the state of automatic writing of data at the end of each sweep:

0 Disable writing to disk at the end of each sweep
1 Enable writing to disk at the end of each sweep

Returns The state of automatic writing to file at the end of each sweep: 0 for disabled, 1 for enabled.

See also: `SampleClear(), SamplePause(), SampleSweep(), SampleStatus()`

SampleZeroOffset()

This function sets and gets the offset of zero on the x-axis for newly sampled data. Normally zero will be at the start of the frame or the trigger point for peri-triggered data.

```
Func SampleZeroOffset({offset});
```

offset If present this sets the amount in seconds by which to offset zero:

Returns The offset in seconds of zero before the function was called.

See also: `SampleClear()`

ScriptBar()

This controls the Script toolbar. Call the command with no arguments to return the number of toolbar buttons. The first button is numbered 0.

```
Func ScriptBar({nBut%{, &get$}});
Func ScriptBar(set$);
```

`nBut%` Set -1 and omit `get$` to clear all buttons and return 0. Otherwise it is a button number and returns -1 if the button does not exist, 0 if it is the last button, and 1 if higher-numbered buttons exist. `get$` returns the information as for `set$`.

`set$` This holds up to 8 characters of button label, a vertical bar, the path to the script file including `.sgs`, a vertical bar and a pop-up comment. The function returns the new number of buttons or -1 if all buttons are already used.

Returns See the descriptions above. Negative return values indicate an error.

For example, the following code clears the script bar and sets a button:

```
ScriptBar(-1);    'clear all buttons
ScriptBar("ToolMake|C:\\Scripts\\ToolMake.sgs|Build a toolbar");
```

See also: `App()`

ScriptRun()

This sets the name of a script to run when the current script terminates. You can pass information to the new script using disk files or by using the `Profile()` command. You can call this function as often as you like; only the last use has any effect.

```
Proc ScriptRun(name${, flags%});
```

`name$` The script file to run. You can supply a path relative to the current folder or a full path to the script file. If you supply a relative path, it must still be valid at the end of the current script. Set `name$` to `""` to cancel running a script.

`flags%` Optional flags that control the new script. If omitted, 0 is used. The only flag defined now is 1 = run new script even if the current script ends in an error.

If the file you name does not exist when Signal tries to run it, nothing happens. If the nominated script is not already loaded, Signal will load it, run it and unload it.

See also: `App()`, `Profile()`

Seconds()

This sets or gets the timer in seconds and is used for relative time measurements. If you want the position reached in the current sweep, use the `Maxtime()` function.

```
Func Seconds({set});
```

`set` If present, this sets the time in seconds.

Returns The time in seconds. If you have not used the set call, the time is the time since the Signal system was started. If the `set` argument is present in the call, the value returned is the value before the new time was set.

See also: `MaxTime()`

Selection$()

This function returns the text in the current view that is currently selected, that is, the text that would be copied to the clipboard if the Edit menu Copy command was used.

```
Func Selection$();
```

Returns The current text selection. If there is no text selected, or if the view is inappropriate for this action, an empty string is returned.

SerialClose()

This function closes a serial port opened by `SerialOpen()`. Closing a port releases memory and system resources. Ports are automatically closed when a script ends, however it is good practice to close a port when your script has finished with it.

```
Func SerialClose(port%);
```

`port%` The serial port to close as defined for `SerialOpen()`.

Returns 0 or a negative error code.

See also: `SerialOpen()`, `SerialWrite()`, `SerialRead()`, `SerialCount()`

SerialCount()

This counts the characters or items buffered in a serial port opened by `SerialOpen()`. Use this to detect input so your script can do other tasks while waiting for serial data. There is an internal buffer (belonging to Signal) of 1024 characters per port that is filled when you use `SerialCount`. The size of this buffer limits the number of characters that this function can tell you about. To avoid character loss when you are not using a serial line handshake, do not buffer up more than a few hundred characters with `SerialCount()`.

```
Func SerialCount(port% {,term$});
```

`port%` The serial port to use as defined for `SerialOpen()`.

`term$` An optional string holding the character(s) that terminate an input item.

Returns If `term$` is absent or empty, this returns the number of characters that could be read. If `term$` is set, this returns the number of complete items that end with `term$` that could be read.

See also: `SerialOpen(), SerialWrite(), SerialRead(), SerialClose()`

SerialOpen()

This function opens a serial port and configures it for use by the other serial line functions. It is not an error to call `SerialOpen()` more than once on the same port. The serial routines use the host operating system serial line support. Consult your system documentation for information on serial line connections and Baud rates limit.

```
Func SerialOpen(port%{, baud%{, bits%{, par%{, stop%{, hsk%}}}}});
```

`port%` The serial port to use, in the range 1 to 9. The number of ports depends on the computer. Two ports (1 and 2) are common on both PC and Macintosh systems.

`baud%` This sets the serial line Baud rate (number of bits per second). The maximum character transfer rate is of order one-tenth this figure. All standard rates from 50 to 115200 Baud are supported. If you do not supply a Baud rate, 9600 is used.

`bits%` The number of data bits used to encode a character. Windows supports 4 to 8 bits, the Macintosh supports 7 or 8. If `bits%` is omitted, 8 is set. Apart from very specialised use, standard values are 7 or 8 data bits. If you set 7 data bits, character codes from 0 to 127 can be read. If you set 8 data bits, codes from 0 to 255 are possible.

`par%` Set this to 0 for no parity check, 1 for odd parity or 2 for even parity. If you do not specify this argument, no parity is set.

`stop%` This sets the number of stop bits as 1 or 2. If omitted, 1 stop bit is set. If you specify 5 data bits, a request for 2 stop bits results in 1.5 stop bits being used.

`hsk%` This sets the handshake mode, sometimes called “flow control”. 0 sets no handshake, 1 sets a hardware handshake, 2 sets XON/XOFF protocol.

Returns 0 or a negative error code.

See also: `SerialWrite(), SerialRead(), SerialCount(), SerialClose()`

SerialRead()

This function reads characters, a string, an array of strings, or binary data from a nominated serial port that was previously opened with `SerialOpen()`. Binary data can include character code 0, string data never includes character 0.

```
Func SerialRead(port%, &in$|in$[]|&in%|in%[]{,term${, max%}});
```

`port%` The serial port to read from as defined for `SerialOpen()`.

`in$` A single string or an array of strings to fill with characters. There is no point providing an array of strings unless you have set a terminator, as without a terminator all input goes to the first string in the array.

`in%` A single integer (`term$` and `max%` are ignored) or an array of integers (`term$` and `max%` can be used) to read binary data. Each integer can hold one character, coded as 0 up to 255. The function returns the number of characters returned.

`term$` If this is an empty string or omitted, all characters read are input to the string, integer array or to the first string in the string array, The number of characters read can be limited by `max%`. The function returns the number of characters read.

If `term$` is not empty, the contents are used to separate data items in the input stream. Only complete items are returned and the terminator is not included. For example, set the terminator to `"\n"` if lines end in line feed, or to `"\r\n"` if input lines end with carriage return then line feed. If `in$` is a string, one item at most is returned. If `in$[]` is an array, one item is returned per array element. The function returns the number of items read unless `in` is an integer, in which case the function returns the number of characters returned.

`max%` If present, it sets the maximum number of characters to read into each string or into the integer array. If a terminator is set, but not found after this many characters, the function breaks the input at this point as if a terminator had been found. There is a maximum limit set by the size of the buffers used by Signal to process data and by the size of the system buffers used outside Signal. This is typically 1024 characters.

Returns The function returns the number of characters or items read or a negative error code. If there is nothing to read, it waits 1 second for characters to arrive before timing out and returning 0. To avoid hanging up Signal, use `SerialCount()` to test for items to read.

See also: `SerialOpen(), SerialWrite(), SerialCount(), SerialClose()`

SerialWrite()

This writes one or more strings, or binary data, to a serial port opened by `SerialOpen()`.

```
Func SerialWrite(port%, out$|out$[]|out%|out%[]{, term$});
```

`port%` The serial port to write to as defined for `SerialOpen()`.

`out$` A single string or an array of strings to write to the output.

`out%` A single integer to write as binary. The maximum value writeable depends on the number of data bits set for the port; 7-bit data writes as `out% band 127`, 8-bit data writes as `out% band 255`.

`out%[]` An array of characters to write, stored one per integer.

`term$` If present, it is written to the output port after the contents of `out%`, `out%[]` or `out$` or after each string in `out$[]`.

Returns The number of strings or binary characters written or a negative error code. If the output system becomes full, the function waits for one second before timing out. If a time-out occurs, the function returns the number of strings sent before the time-out.

See also: `SerialOpen(), SerialRead(), SerialCount(), SerialClose()`

SetXXX() commands

This family of commands creates memory view windows. Memory views which require processing are derived from and attached to the current data view, which should be a file view. This does not apply to `SetCopy()` and `SetMemory()`, which create memory views that do not use a `Process` command. The `SetTrend()` function is special in that it creates an XY view which will receive data points from processing a source view which can be either a file or a memory view. The `SetXXX()` functions do not update the display, for which you should use `Draw()` or `DrawAll()`.

All these functions, with the exception of `SetOpCl()` return a positive view handle if they succeed or a negative error code. Possible errors are: bad channel number, illegal number of bins and out of memory. The new derived memory view will be empty until a processing command is executed for it. The processing of a memory view takes data from the source view and replaces or adds to the data in the memory view. `SetOpCl()` creates a idealised trace channel in the source view to hold an idealised trace and returns the channel number.

When these functions create a new view, it is made the current view. The view is created invisibly and must be made visible with `WindowVisible(1)` before it will appear.

See also: `Process(), ProcessAll(), ProcessFrames(), SetAutoAv(), SetAverage(), SetLeak(), SetOpCl(), SetOpClScan(), SetPower(), SetTrend(), SetCopy(), SetMemory()`

SetAmplitude()

This function creates a memory view to hold an amplitude histogram in each channel when it is processed. `Sweeps()` reports the number of sweeps of waveform data accumulated by processing into the memory view. The current view when `SetAmplitude()` is called will be the source view for the data to be processed. In this version of Signal the source view cannot be a memory view.

```
Func SetAmplitude(ch%, bins% {, minAmp|minAmp${, maxAmp|maxAmp$
                           {, sTime|sTime${, eTime|eTime$}}});
```

`ch%` A waveform channel to analyse from the current view. Use a channel number (1 to n).

`bins%` The number of bins in the resulting histogram.

`minAmp` The smallest amplitude to be represented in the histogram. The default is the lower limit of the ADC range of the channel.

`minAmp$` The smallest amplitude to be represented in the histogram, as a string. Strings such as “Hcursor(1)” can be used.

`maxAmp` The largest amplitude to be represented in the histogram. The default is the upper limit of the ADC range of the channel.

`maxAmp$` The largest amplitude to be represented in the histogram, as a string. Strings such as “Hcursor(1)” can be used.

`sTime` The start time of the data to be included in the analysis. The default is the minimum time in the frame.

`sTime$` A sting giving the start time of the data to be included in the analysis, e.g. “Cursor(1)”.

`eTime` The end time of the data to be included in the analysis. The default is the maximum time in the frame.

`eTime$` A sting giving the end time of the data to be included in the analysis e.g. “Cursor(2)”.

Returns The function returns a handle for the new view, or a negative error code.

See also: `SetXXX(), Process(), ProcessAll(), ProcessFrames(), Sweeps(), View()`

SetAutoAv()

This function creates a memory view to hold a sum or average in each channel when it is processed. The memory view will hold multiple frames, with set numbers of source frames being averaged into each destination frame. This allows you to set up averaging with, for example, every ten source frames processed into a new average. The amount moved-on between averages can be separately controlled for extra flexibility. The current view when `SetAutoAv()` is called will be the source view for the data to be processed. In this version of Signal the source view can not be a memory view. `SetAutoAv()` is very similar to `SetAverage()`.

```
Func SetAutoAv(cSpc, perAv%, betAv%{, width, offs
                    {, sum%{ ,xzero%{, cntExc%{, doErrs%}}}}});
```

`cSpc` A channel list specifier of the channels to average. See the *Script language syntax* chapter for a definition of channel specifiers.

`perAv%` The number of source frames to use per average.

`betAv%` The number of source frames between the first frame for one average and the first frame for the next. If `perAv%` is the same as `betAv%`, then each `perAv%` frames processed make a new average frame. If `betAv%` is less than `perAv%`, then some source frames are used for more than one average; if it is greater than `perAv%` then some source frames will be unused.

`width` The width of the average in x axis units. If omitted the whole frame will be used. The maximum is limited by available memory.

`offs` This sets the offset in x axis units from start of frame to the start of the data to average. If omitted or zero, the data will be taken from the start of the frame.

`sum%` If present and non-zero, each channel in the memory view will hold the sum of the data accumulated. If omitted or zero, the memory view channels will hold the mean of the data accumulated.

`xzero%` If present and non-zero, this forces the x axis of the memory view to start at zero. If omitted or zero, the start of the x axis will be the same as the start of the data that is averaged.

`cntExc%` If present and non-zero, excluded frames will count as if they had been added, so Signal will not continue to search for enough frames to form each average, and so remain in step with the sampling protocol.

`doErrs%` If present and set to 1 then error bar information will be generated.

Returns The function returns a handle for the new view, or a negative error code.

See also: `SetXXX(), SetAverage(), Process(), ProcessAll(), ProcessFrames(), Sweeps(), View(), MinTime()`

SetAverage()

This function creates a memory view to hold a sum or average in each channel when it is processed. `Sweeps()` reports the number of sweeps of waveform data accumulated by processing into the memory view. The current view when `SetAverage()` is called will be the source view for the data to be processed. In this version of Signal the source view cannot be a memory view.

```
Func SetAverage(cSpc{, width, offs{, sum%{, xzero%{, doErrs%}}}});
```

`cSpc` A channel list specifier of the channels to average. See the *Script language syntax* chapter for a definition of channel specifiers.

`width` The width of the average in x axis units. If omitted the whole frame will be used. The maximum is limited by available memory.

`offs` This sets the offset in x axis units from start of frame to the start of the data to average. If omitted or zero, the data will be taken from the start of the frame.

`sum%` If present and non-zero, each channel in the memory view will hold the sum of the data accumulated. If omitted or zero, the memory view channels will hold the mean of the data accumulated.

`xzero%` If present and non-zero, this forces the x axis of the memory view to start at zero. If omitted or zero, the start of the x axis will be the same as the start of the data to average, as defined by offset `offs` from `MinTime()` in the current frame.

`doErrs%` If present and set to 1 then error bar information will be generated.

Returns The function returns a handle for the new view, or a negative error code.

See also: `SetXXX(), SetAutoAv(), Process(), ProcessAll(), ProcessFrames(), Sweeps(), View(), MinTime()`

SetCopy()

This function creates a new memory view with channels selected from and identical to those in the current view. The new view can be empty or contain data copied from the current frame. It is attached to no source view and has no implied `Process()`.

```
Func SetCopy(cSpc, title$, bcopy%);
```

`cSpc` A channel list specifier of the channels to copy. See the *Script language syntax* chapter for a definition of channel specifiers.

`title$` The new window title.

`bcopy%` If this is not 0 the data values are copied into the new memory view. If this is 0 the waveform data values in the new view are zero and marker channels are empty.

Returns A handle for the new view, or a negative error code.

See also: `SetXXX(), SetMemory(), View()`

SetLeak()

This function creates a memory view to hold leak subtracted data when it is processed. The current view when `SetLeak()` is called, which must be a file view, will be the source view for the data to be processed.

```
Func SetLeak(mode%, chan%, stim%, base|base$, pulse|pulse$, width,
                                  form%, sub%{, zero%{, cntExc%}});
```

`mode%` A value to set the leak subtraction mode: 0 for Basic, 1 for P/N or 2 for States.

`chan%` A single waveform channel from the current view, this is the channel that will be leak-subtracted, all other source channels are copied unchanged. Use a channel number (1 to n).

`stim%` A single waveform channel from the current view. This is the channel that will be used to measure the stimulus pulse size. Normally this will be a channel on which the stimulus was recorded.

`base` A time at which the baseline level can be measured; a time outside the stimulus pulse.

`base$` The baseline level time expressed as a string, allowing constructs such as `"Cursor(1) - 10"`.

`pulse` A time at which the pulse level can be measured; a time inside the stimulus pulse.

`pulse$` The pulse time expressed as a string.

`width` The width of the two level measurements. The measurement used is the average of all waveform points within the specified width.

`form%` The first frame used to measure the leak in Basic mode, the number of frames to use for the leak in P/N mode and the state code for leak frames in States mode.

`sub%` The last frame used to measure the leak in Basic mode and the number of frames to subtract the current leak from in P/N mode. This parameter is unused in States mode.

`zero%` If present and non-zero, the baseline level will be maintained constant by the leak subtraction process; otherwise this adjustment is not done.

`cntExc%` If present and non-zero, excluded frames will count as if they had been used, so Signal will not continue to search for enough frames to form each leak and so remain in step with the sampling protocol.

Returns The function returns a handle for the new view, or a negative error code.

See also: `SetXXX(), FrameState(), SetPower(), Process(), ProcessAll(), ProcessFrames(), View()`

SetMemory()

This function creates a memory view of user-defined type, attached to no source view and with no implied `Process()`.

```
Func SetMemory(chans%, pts%, binsz, offset, marks%, tmks%,
                          mkBns%, title$, xU${, yU${, xT${, yT$}}});
```

`chans%` The number of waveform channels in the view (1 to 80).

`pts%` The number of data points in each waveform channel.

`binsz` The x axis increment per point in the waveform channels. This is equivalent to the sample interval for sampled data. This value should be positive and non-zero.

`offset` The x axis value at the first point of the waveform channels.

`marks%` The number of marker channels in the view, not including text markers (0 to 80).

`tmks%` The number of text marker channels in the view. Not implemented yet.

`mkBns%` The number of marker items in each marker channel.

`title$` The new window title.

`xU$` The x axis units.

`yU$` Optional, y axis units, blank if omitted.

`xT$` Optional, x axis title (otherwise blank).

`yT$` Optional, y axis title (otherwise blank).

Returns The function returns a handle for the new view, or a negative error code.

See also: `SetXXX()`, `SetCopy()`, `View()`

SetOpCl()

This function creates an idealised trace process. The current view when `SetOpCl()` is called will be the source view for the data to be processed.

```
Func SetOpCl(cSpc, start|start$, end|end$, level1|level1$,
             level2|level2$, base|base${, interp%{, track%{,
             flags%}}});
```

`cSpc` A channel list specifier of the channels to analyse. See the *Script language syntax* chapter for a definition of channel specifiers.

`start` The start time in seconds of the idealised trace to be created.

`start$` The start time as a string, e.g. "`Cursor(1)`".

`end` The end time in seconds of the idealised trace to be created.

`end$` The end time as a string, e.g. "`Cursor(2)`".

`level1` The level which the original data trace must cross in order to change from closed state to open state.

`level1$` The level1 value as a string, e.g. "`HCursor(1)`".

`level2` The level which the original data trace must cross in order to change from open state to closed state.

`level2$` The level2 value as a string, e.g. "`HCursor(2)`".

`base` The level at which the data trace is considered to be in the closed state. This is used for calculating additional thresholds for multi-level data.

`base$` The base level as a string, e.g. "`HCursor(3)`".

`interp%` The interpolation method to use for calculating the time of a threshold crossing. Set to 0 for no interpolation and 1 for linear interpolation. The default value is 0.

`track%` The number of data points in the closed state to use in tracking the base level, in order to adjust the thresholds to compensate for baseline drift. The default value is 0.

`flags%` A set of flags built up by adding together the following values:

0x0001 1 Outward current. An opening of a channel leads to a more positive current.

0x0002 2 Multiple level data. Normally set if there is more than one channel in the patch.

This parameter is set to 0 by default.

Returns The function returns the number of the idealised trace channel.

See also: `SetXXX(), SetOpClAmp(), SetOpClBurst(), SetOpClHist(), SetOpClScan()`

SetOpClAmp()

This function creates a memory view to hold an open/closed amplitude histogram from an idealised trace. The current view when `SetOpClAmp()` is called will be the source view for the data to be processed.

```
Func SetOpClAmp(chan%, bins%, minAmp|minAmp$, maxAmp|maxAmp$,
                incl%, excl%);
```

`chan%` The channel number in the source view. This channel must have an idealised trace fitted, for a histogram to be built.

`bins%` The number of bins in the resulting histogram.

`minAmp` The smallest amplitude to be represented in the histogram.

`minAmp$` The smallest amplitude to be represented in the histogram, as a string. Strings such as “Hcursor(1)” can be used.

`maxAmp` The largest amplitude to be represented in the histogram.

`maxAmp$` The largest amplitude to be represented in the histogram, as a string. Strings such as “Hcursor(1)” can be used.

`incl%` A set of flags associated with each open/closed time to include. If an event has any of the flags in the `incl%` set and none of the flags in the `excl%` set, it will be included in the histogram. See `SetOpClHist()` for a list of flag values.

`excl%` A set of flags defining those events to be excluded from the histogram. Events having flags in the `excl%` set will be excluded regardless of whether they have flags in the `incl%` sets.

Returns The function returns a handle for the new view, or a negative error code.

See also: `SetXXX(), SetOpCl(), SetOpClBurst(), SetOpClHist(), SetOpClScan()`

SetOpClBurst()

This function creates a memory view to hold an open/closed burst time histogram from an idealised trace. The current view when `SetOpClBurst()` is called will be the source view for the data to be processed. A burst duration is from the start time of an included event to the start time of an excluded event having a duration greater than the critical interval.

```
Func SetOpClBurst(chan%, binsz, maxDur, crInt, incl%, excl%);
```

`chan%` The channel number in the source view. This channel must have an idealised trace fitted for a histogram to be built.

`binsz` The x increment per bin in the histogram.

`maxDur` The maximum duration of a burst to be included in the histogram.

`crInt` The critical interval.

`incl%` A set of flags associated with each open/closed time to include. If an event has any of the flags in the `incl%` set and none of the flags in the `excl%` set, it will be included in the histogram. See `SetOpClHist()` for a list of flag values.

`excl%` A set of flags defining those events to be excluded from the histogram. Events having flags in the `excl%` set will be excluded regardless of whether they have flags in the `incl%` sets.

Returns The function returns a handle for the new view, or a negative error code.

See also: `SetXXX()`, `SetOpCl()`, `SetOpClAmp()`, `SetOpClHist()`, `SetOpClScan()`

SetOpClHist()

This function creates a memory view to hold an open/closed time histogram from an idealised trace. The current view when `SetOpClHist()` is called will be the source view for the data to be processed.

```
Func SetOpClHist(chan%, binsz, maxDur, incl%, excl%);
```

`chan%` The channel number in the source view. This channel must have an idealised trace fitted for a histogram to be built.

`binsz` The x increment per bin in the histogram.

`maxDur` The maximum duration of an open/closed time to be included in the histogram.

`incl%` A set of flags associated with each open/closed time to include. If an event has any of the flags in the `incl%` set and none of the flags in the `excl%` set it will be included in the histogram. A set of flags is built up by adding together the folowing values:

0x0001	1	Level 1. Closed times and the first open level will have this set.
0x0002	2	Bad data: Events marked as bad in the idealised trace editor.
0x0004	4	Assumed amplitude: Events whose amplitude has not been calculated from the raw data.
0x0008	8	Spare.
0x0010	16	First latency: The period from the start of the idealised trace to the first transition.
0x0020	32	Truncated: The last event in an idealised trace.
0x0040	64	Closed time.
0x0080	128	Open time.
0x0100	256	Spare.
0x0200	512	Spare.
0x0400	1024	Spare.
0x0800	2048	Level 6 of multi-level data.
0x1000	4096	Level 5 of multi-level data.
0x2000	8192	Level 4 of multi-level data.
0x4000	16384	Level 3 of multi-level data.
0x8000	32768	Level 2 of multi-level data.

`excl%` A set of flags defining those events to be excluded from the histogram. Events having flags in both the `incl%` and `excl%` sets will be excluded.

Returns The function returns a handle for the new view, or a negative error code.

See also: `SetXXX(), SetOpCl(), SetOpClAmp(), SetOpClBurst(), SetOpClScan()`

SetOpClScan()

This function creates an idealised trace process using the SCAN method. The current view when `SetOpClScan()` is called will be the source view for the data to be processed.

```
Func SetOpClScan(cSpc, start|start$, end|end$, base|base$,
              thresh|thresh$, open|open$, cutoff{, track%{,
              flags%}});
```

`cSpc` A channel list specifier of the channels to analyse. See the *Script language syntax* chapter for a definition of channel specifiers.

`start` The start time in seconds of the idealised trace to be created.

`start$` The start time as a string, e.g. `"Cursor(1)"`.

`end` The end time in seconds of the idealised trace to be created.

`end$` The end time as a string, e.g. `"Cursor(2)"`.

`base` The initial level of the baseline.

`base$` The `base` value as a string, e.g. `"HCursor(1)"`.

`thresh` The level which the original data trace must cross in order to change from closed state to an open state.

`thresh$` The `thresh` value as a string e.g. `"HCursor(2)"`.

`open` The approximate initial level at which the channel will be assummed to be fully open.

`open$` The `open` value as a string, e.g. `"HCursor(3)"`.

`cutoff` The –3dB frequency in Hz of the cut-off frequency of the filter used to remove noise from the raw data..

`track%` The number of data points over which to form a running average of the baseline. The default value is 0.

`flags%` A set of flags built up by adding together the following values:

1 Outward current. An opening of a channel leads to a more positive current.

2 Avoid sub-levels. Normally set if you know there are no sub-levels so multiple transitions will be used instead where possible.

This parameter is set to 0 by default.

Returns The function returns the number of the idealised trace channel.

See also: `SetXXX()`, `SetOpClAmp()`, `SetOpClBurst()`, `SetOpClHist()`, `SetOpCl()`

SetPower()

This function creates a memory view to hold a power spectrum in each channel when it is processed. `Sweeps()` reports the number of sweeps accumulated by processing into the memory view. The current view when `SetPower()` is called will be the source view for the data to be processed. In this version of Signal the source view cannot be a memory view.

```
Func SetPower(cSpc, fftsz%{, offset});
```

`cSpc` A channel list specifier of the channels to analyse. See the *Script language syntax* chapter for a definition of channel specifiers.

`fftsz%` The size of the transform used in the FFT. This must be a power of 2 in the range 16 to 4096. The memory view has half this number of bins. The width of each bin is the sampling rate of the channel divided by `fftsz%`. Each block of `fftsz%` data points processed increments the value for `Sweeps()`.

`offset` This sets the offset in x axis units from start of frame to the start of the data to analyse. If omitted or zero, the data will be taken from the start of the frame.

Returns The function returns a handle for the new view, or a negative error code.

See also: `SetXXX(), SetAverage(), ArrFFT(), Process(), Sweeps(), View()`

SetTrend()

This function creates an XY view to hold XY points calculated by processing. The current view when `SetTrend()` is called, which can be a file or memory view, will be the source view for the data to be processed. The `SetTrend` function creates an XY view with a single channel; more channels can be added using `SetTrendChan()`.

```
Func SetTrend(name$, xtyp%, xc%, xp|xp$, xb|xb$, xw, ytyp%, yc%,
                 yp|yp$, yb|yb$, yw {, pts%{, xFitCh%{, yFitCh%{,
                 comX%}}}});
```

`name$` The name of the channel. The channel name is shown in the XY key area.

`xtyp%` The type of measurement to take for the X part of each point. The possible values are:

100	Value at a point.
101	Value difference between points.
102	Time at a point.
103	Time difference between points.
104	Frame number.
105	Absolute time of frame.
106	Frame state value.
107	Fit coefficient.
108	User entered value
109	Value ratio.
110	Value product.
111	Value above baseline.
1	Curve area between points.
2	Mean between points.
3	Slope between points.
4	Area between points.
5	Sum between points.
6	Modulus between points.
7	Maximum value between points.
8	Minimum value between points.
9	Amplitude value between points.
10	RMS Amplitude value between points.
11	Standard deviation value between points.

12 Absolute maximum value between points.
13 Peak found between points.
14 Trough found between points.

Please note: These values have been changed in Signal version 3.10; any scripts used with earlier versions of Signal will need adjustment to use the new values.

`xc%` A single waveform channel from the current view, this is the channel that will be used to take the X measurement. Use a channel number (1 to n). For `xtyp%` = 107 this is the fit coefficient index.

`xp` The time for single-point X measurements and difference measurements. For measurements between points, this is the end time.

`xp$` The time for single-point X measurements and difference measurements expressed as a string. This allows constructs such as "`Cursor(1) - 10`", to be used.

`xb` The reference time for difference measurements; for measurements between points, this is the start time.

`xb$` The reference time for difference measurements expressed as a string.

`xw` The width used for some measurements, particularly value at point and value difference.

`ytyp%` The type of measurement to take for the Y part of each point. The possible values are the same as for `xtyp%`.

`yc%` A single waveform channel from the current view; this is the channel that will be used to take the Y measurement. Use a channel number (1 to n). For `ytyp%` = 7 this is the coefficient index.

`yp` The time for single-point Y measurements and difference measurements. For measurements between points, this is the end time.

`yp$` The time for single-point Y measurements and difference measurements expressed as a string. This allows constructs such as "`Cursor(1) - 10`", to be used.

`yb` The reference time for difference measurements; for measurements between points, this is the start time.

`yb$` The reference time for difference measurements expressed as a string.

`yw` The width used for some measurements, particularly value at point and value difference.

`pts%` The number of points for this channel before they are recycled. If this is omitted or set to zero, all points are simply added.

`xFitCh%` The channel containing the fit to take fit coefficient values from if `xtyp%` = 107. If this is set to 0 (the defauilt) Signal will scan all channels to find one containing a fit with sufficient coeffiecients to use.

`yFitCh%` The channel containing the fit to take fiit coefficient values from from if `ytyp% =` 107. If this is set to 0 (the defauilt) Signal will scan all channels to find one containing a fit with sufficient coeffiecients to use.

`comX%` Set to 1 for all channels in the plot to share the same x-values. 0 (default) sets channels to have independent x-values.

Returns The function returns a handle for the new view, or a negative error code.

See also: `SetXXX(), FrameState(), FrameAbsStart(), ChanMeasure(), SetTrendChan(), Process(), ProcessAll(), ProcessFrames()`

SetTrendChan()

This function adds another channel to an XY view created using `SetTrend()`. The current view when `SetTrendChan()` is called must be the XY view to be modified. The `SetTrendChan()` function can be used to create XY views with up to 32 channels.

```
Func SetTrendChan(name$, xtyp%, xc%, xp|xp$, xb|xb$, xw, ytyp%,
          yc%, yp|yp$, yb|yb$, yw {, pts%{, xFitCh%{, yFitCh%}}});
```

All of the parameters to `SetTrendChan()` are exactly the same as for `SetTrend()`.

Returns The function returns zero or a negative error code.

See also: `SetXXX(), FrameState(), FrameAbsStart(), ChanMeasure(), SetTrend(), Process(), ProcessAll(), ProcessFrames()`

ShowBuffer()

This function gets or sets the show frame buffer flag from the current view.

```
Func ShowBuffer({yes%});
```

`yes%` If this is non-zero the frame buffer is shown otherwise the current frame data is shown. If this is omitted, no change is made.

Returns The buffer show flag at the time of the call.

See also: `BuffXXX(), Frame()`

ShowFunc()

This function draws a function over a data channel. This function is included for compatibility. It has been replaced with `ChanFitShow()` in version 3 or later.

```
Func ShowFunc(func%, chan% {, start, coefs[]});
```

`func%` The type of the function to show:

0 Don't show a function
1 Single exponential
2 Double exponential
3 Single gaussian
4 Double gaussian

`chan%` The channel on which to show the function.

`start` The time to start drawing from.

`coefs` The coefficients to use in drawing the function.

Returns The function returns zero or a negative error code.

See also: `FitExp(), FitGauss(), ChanFitShow()`

Sin()

This function calculates the sine of an angle in radians, or converts an array of angles into an array of sines.

```
Func Sin(x|x[]{[]...});
```

`x` The angle, expressed in radians, or a real array of angles. The best accuracy of the result is obtained when the angle is in the range -2π to 2π.

Returns When the argument is an array, the function replaces the array with the sines of all the points and returns either a negative error code or 0 if all was well. When the argument is not an array the function returns the sine of the angle.

See also: `ATan(), Cos(), Cosh(), Ln(), Log(), Pow(),  Sinh(), Sqrt(),Tan()`

Sinh()

This calculates the hyperbolic sine of a value or of an array of values.

`Func Sinh(x|x[]{[]...});`

`x`	The value, or an array of real values.
Returns	When the argument is an array, the function replaces the array elements with their hyperbolic sines and returns 0. When the argument is not an array the function returns the hyperbolic sine of the argument.

See also: `Abs()`, `ATan()`, `Cos()`, `Cosh()`, `Exp()`, `Frac()`, `Ln()`, `Log()`, `Max()`, `Min()`, `Pow()`, `Rand()`, `Round()`, `Sqrt()`, `Sinh()`, `Tan()`, `Tanh()`, `Trunc()`

Sound()

This has two variants. The first plays a tone of set pitch and duration in Windows NT/XP and a short "beep" in Windows 98/ME. The second plays a `.WAV` file or system sound if your system has multimedia support. The `.WAV` output was added at version 2.05.

`Func Sound(freq%, dur{, midi%});` Tone output
`Func Sound(name${, flags%});` Multimedia sound output

`freq%` If `midi%` is 0 or omitted this holds the sound frequency in Hz. If `midi%` is non-zero this is a MIDI value in the range 1-127. A MIDI value of 60 is middle C, 61 is C# and so on. Add or subtract 12 to change the note by one octave.

`dur` The sound duration, in seconds. The script stops during output.

`midi%` If this is present and non-zero, the frequency is interpreted as a MIDI value, otherwise it is a frequency in Hz.

`name$` Either the name of `.wav` file or the name of a system sound. You can either supply the full path to the file or just a file name and the system will search for the file in the current directory, the Windows directory, the Windows system directory, directories listed in the `PATH` environmental variable and the list of directories mapped in a network. If no file extension is given, `.wav` is assumed. The file must be short enough to fit in available physical memory, so this function is suitable for files of a few seconds duration only.

A blank name halts any current sound output. If `name$` is any of the following (case is important), a standard system sound plays:

`"S*"`	Asterisk	`"SS"`	System start	`"SE"`	System exit
`"S?"`	Query	`"SW"`	System welcome	`"SD"`	System default
`"SH"`	Hand	`"S!"`	System exclamation		

`flags%` This optional argument controls how the data is played. It is the sum of the following values (given in hexadecimal and decimal):

0x0001	1	Play asynchronously (start output and return). Without this flag, Signal does nothing (including sampling) until replay ends.
0x0002	2	Silence when sound not found. Normally `Sound()` plays the system default sound if the nominated sound cannot be found.
0x0008	8	Loop sound until stopped by another `Sound()` command. You must also supply the asynchronous flag if you use loop mode.
0x0010	16	Don't stop a playing sound. Normally, unless the "No wait" flag is set, each command cancels any playing sound.
0x2000	8192	No wait if sound is already playing. `Sound("",0x2010)` can be used to detect if a previous asynchronous sound has finished.

If you don't supply this argument, the flag value is set to 0x2000.

Returns The tone output returns 0 or a negative error code. The multimedia output returns non-zero if the function succeeded and zero if it failed.

See also: `Speak()`

Speak()

If your system supports text to speech, this command allows you to convert a text string into speech. Currently we provide no facilities to setup voices or to route the sound output; you must do this from the Speech applet in the control panel.

`Func Speak(text${, opt%});` Convert text to speech

`Func Speak({what%{, val}});` Speech status and control

`text$` A string holding the text to output, for example `"Sampling has started."`.

`opt%` This optional argument (default value 1) controls the text conversion and output method. It is the sum of the following flags:

- 1 Speak asynchronously. Without this flag the command waits until speech output is over before returning.
- 2 Cancel any pending speech output.
- 4 Speak punctuation marks in the text.
- 8 Process embedded SAPI XML; the www.microsoft.com/speech web site has more information on this advanced topic. For example:
  `Speak("Emphasis on <EMPH>this</EMPH> word",8);`
- 16 Reset to the standard voice settings before speaking.

`what%` An optional variable, taken as 0 if it is omitted:

- 0 Returns 1 if speech is playing and 0 if it is not.
- 1 Wait for up to `val` seconds (default value 3.0) for output to end. The return value is 0 if playing is finished, 1 if it continues after `val` seconds.
- 2 Returns the current speech speed in the range –10 to 10; 0 is the standard speed. If `val` is present, it sets the new speed.
- 3 Returns the current speech volume in the range 0 to 100, 100 is the standard volume. If `val` is present, it sets the new volume.

`val` An optional argument used when `what%` is greater than 0.

Returns If there is no speech support available, or a system error occurs, the command returns -1. Otherwise the first command variant returns 0 if all is well and the second variant returns the values listed for `what%`.

To use TTS (text to speech), you need a suitable sound card and the Microsoft SAPI software support. Windows XP has this software included with the operating system. You can get text to speech support as a download for other versions of windows. In August 2004, the speech support was available as `SpeechSDK51MSM.exe` from the web page `www.microsoft.com/speech/download/sdk51/` but this location may change.

See also: `Sound()`

Sqrt()

Forms the square root of a real number or replace each element of an array of real numbers with their sqaure roots. Negative numbers cause the script to halt with an error when x is not an array. With an array, negative numbers are set to 0 and an error is returned.

`Func Sqrt(x|x[]{[]...});`

`x` A real number or a real array to replace with an array of square roots.

Returns With an array, this returns 0 if all was well, or a negative error code. With an expression, it returns the square root of the expression.

Str$()

This converts a number to a string.

```
Func Str$(x {,width% {,decs%}});
```

`x` A number to be converted.

`width%` Optional minimum field width. The number is right justified in this width.

`decs%` Optional number of decimal places.

Returns A string holding a representation of the number.

See also: `Print$(), Print(), Val()`

Sweeps()

This function returns the number of sweeps accumulated into the frame data. If the memory view is saved and reloaded as a file view, the sweeps value is preserved. What each item or sweep is, depends on the type of the analysis.

```
Func Sweeps({new});
```

`new` If present, sets the sweeps value for the frame to a new value. Note that, while setting the sweep count for a frame is necessary in some circumstances, if used incautiously this mechanism will corrupt the sweep count of analysed data.

Returns The number of sweeps accumulated to produce the frame data.

See also: `SetAverage(), SetPower(), View()`

System$()

This returns the operating system name as a string and accesses environment variables belonging to Signal. The environment holds a list of strings of the form `"name=value"`. If you know the name, you can get or set the associated value. You can also read back the entire list of strings into a string array.

```
Func System$({var$ {,value$}});
Func System$(list$[] {,&n%});
```

`var$` If present, this is the name of an environment variable (case insensitive).

`value$` If present, the new value. An empty string deletes the environment variable.

`list$` An array of strings to fill with environment strings of the form `"name=value"`.

`n%` An optional integer that is returned holding the number of elements copied.

Returns With no arguments, it returns: `"Windows SS build n"` where `SS` is the operating system and `n` is the build number. Otherwise it returns the value of the environment variable identified by `var$` or an empty string.

The following example shows how to use this function:

```
var list$[200], value$, n%, i%;
PrintLog("%s\n", System$()); 'Print OS name
System$("fred","good");       'Assign the value good to fred
PrintLog("%s\n", System$("fred")); 'get value of fred
System$("fred","");           'Delete fred from the environment
System$(list$[], n%);         'Print all environment strings
for i%:=0 to n%-1 do PrintLog("%s\n",list$[i%]) next;
```

Each process has its own copy of the environment. Changes you make here only affect the local environment. If you use `ProgRun()` to start another process, it inherits the Signal environment, so you can use the environment to pass information to the new process. However, you cannot see environment changes that the new process makes.

See also: `ProgRun(), System()`

System()

This function returns the operating system version as a number and gets information about desktop screens. Use the `App()` command to get the version number of Signal.

```
Func System({get%{, scr%{, sz%[]}}});
```

`get%` If omitted or 0, the return value is the operating system revision times 100: 351=NT 3.51, 400=95 and NT 4, 410=98, 490=Me, 500=NT 2000, 501=XP, 502=XP x64 Edition, 600=Vista. If 1, the function returns information about installed desktop monitors.

`scr%` Set 0 to return the number of desktop monitors; `sz%[]` gets the pixel co-ordinates of the desktop. Set to n (>0) to get the pixel co-ordinates of screen n; returns 1 for the primary monitor, 0 if not and -1 if it does not exist.

`sz%[]` Optional array of at least 4 elements to return pixel positions. Elements 0 and 1 hold the top left x and y, 2 and 3 hold bottom right x and y.

See also: `App(), System$(), Window(), WindowVisible()`

Tan()

This calculates the tangent of an angle in radians or converts an array of angles into tangents. Tangents of odd multiples of $\pi/2$ are infinite, so cause computational overflow. There are 2π radians in 360°. The value of π is 3.14159265359 (`4.0*ATan(1)`).

```
Func Tan(x|x[]{[]...});
```

`x` The angle, expressed in radians, or a real array of angles. The best accuracy of the result is obtained when the angle is in the range -2π to 2π.

Returns For an array, it returns a negative error code (for overflow) or 0. When the argument is not an array the function returns the tangent of the angle.

See also: `ATan(), Cos(), Cosh(), Ln(), Log(), Pow(), Sin(), Sinh(), Sqrt(), Tanh()`

Tanh()

This calculates the hyperbolic tangent of a value or an array of values.

```
Func Tanh(x|x[]{[]...});
```

`x` The value or an array of real values.

Returns For an array, it returns 0. Otherwise it returns the hyperbolic tangent of `x`.

See also: `ATan(), Cos(), Cosh(), Ln(), Log(), Pow(), Sin(), Sinh(), Sqrt(), Tan()`

Time$()

This function returns the current system time of day as a string. If no arguments are supplied, the returned string shows hours, minutes and seconds in a format determined by the operating system settings. To obtain the time as numbers, use the `TimeDate()` function. To obtain relative time (and fractions of a second), use `Seconds()`. **Warning**: this command does not exist in version 1.00.

```
Func Time$({tBase%, {show%, {amPm%, {sep$}}}});
```

`tBase%` Specifies the time base to show the time in. You can choose between 24 hour clock or 12 hour clock mode. If this argument is omitted, a value of 0 is used.

0 Operating system settings.
1 24 hour format.
2 12 hour format.

`show%` Specifies which time fields to show. Add the values of the required options together and enter that number as the argument. If this argument is omitted or a value of 0 is used, 7 (1+2+4) is used for 24 hour format and 15 (1+2+4+8) for 12 hour format.

1 Show hours.
2 Show minutes.
4 Show seconds.
8 Remove leading zeros from hours.

`amPm%` This sets the position of the "`AM`" or "`PM`" string in 12 hour format. This parameter has no effect in 24 hour format. If this argument is omitted, a value of zero is used. The actual string which gets printed ("`AM`" or "`PM`") is specified by the operating system.

0 Operating system settings.
1 Show to the right of the time.
2 Show to the left of the time.
3 Hide the "`AM`" or "`PM`" string.

`sep$` This string appears between adjacent time fields. If `sep$` = "`:`" then the time will appear as `12:04:45`. If an empty string is entered or `sep$` is omitted, the operating system settings are used.

See also: `Date$()`, `Seconds()`, `TimeDate()`

TimeDate()

This procedure returns the time and date in seconds, minutes, hours, days, months, and years. It can also return the day of the week. You can either enter a separate variable for each field to be returned, or alternatively, an integer array of the desired size. This procedure returns numerical values. If you wish to have a formatted string containing either the date or the time you should use `Date$()` and `Time$()`. If you want to measure relative times, or times to a fraction of a second, see the `Seconds()` command. To get the current sampling time, see `MaxTime()`. **Warning**: this command is not in version 1.00.

```
Proc TimeDate(&s%, {&m%, {&h%, {&d%, {&mon%, {&y%, {&wDay%}}}}}});
Proc TimeDate(now%[]);
```

`s%` If this is the only argument is passed, the number of seconds since midnight is returned in this variable. If the `min%` argument is present, the number of seconds since the beginning of the present minute is returned.

`m%` If this is the last argument, then the number of minutes since midnight is returned in this variable. If `hour%` is present, then the number of full minutes since the beginning of the present hour is returned.

`h%` If present, the number of hours since midnight is returned in this variable.

`d%` If present, the day of the month is returned as an integer in the range 1 to 31.

`mon%` If present, the month number is returned as an integer in the range 1 to 12.

`y%` If present,. the year number is returned here. It will be an integer such as 1998 (the year 2000 is returned as 2000, not 0).

`wDay%` If present, the day of the week will be returned here. This will be an integer in the range 0 (Monday) to 6 (Sunday).

`now%[]` If an array is passed as the first and only arguments, array elements are filled with time and date data. Elements beyond the seventh are not changed. The array can be less than seven elements long. Element 0 is set to the seconds, 1 to the minutes, 2 to the hours, and so on.

See also: `Date$(), MaxTime(), Seconds(), Time$()`

TimeRatio()

This function returns the ratio between the current view X axis units and seconds; for example in milliseconds mode it returns 1000. Use of this value allows script output to use the preferred time units, as the script functions always see time values in seconds, regardless of the time units preferred.

```
Func TimeRatio();
```

Returns The current time ratio.

See also: `TimeUnits$()`

TimeUnits$()

This function returns the current view time units, for example in milliseconds mode it returns “ms”. Use of this allows script output to show the preferred time units.

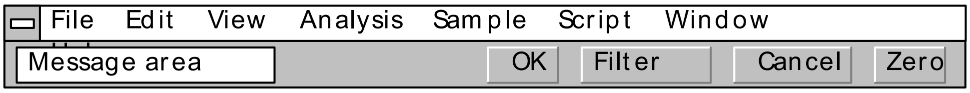
```
Func TimeUnits$();
```

Returns The current time units.

See also: `TimeRatio()`

The toolbar

The toolbar is at the top of the screen, below the menu. The bar has a message area and can hold buttons that are used in the `Interact()` and `Toolbar()` commands.



It is possible to link user-defined functions and procedures to the toolbar buttons. This is done through a set of functions that define buttons and optionally link the buttons to the toolbar. You can define up to 17 buttons in your toolbar, but you will probably be limited by the available space to a maximum of around 10. Buttons are numbered from 1 to 17. There is an invisible button, numbered 0, that is used to set a function that is called when the toolbar is waiting for a button to be pressed.

When you start a script, the toolbar is invisible and contains no buttons. When a script stops running, the toolbar becomes invisible (if it was visible).

Toolbar building

There is an example script `Toolmake.s2s` which automates the writing of toolbar commands to generate your desired toolbar.

See also: `Interact()`, `Toolbar()`, `ToolbarClear()`, `ToolbarEnable()`, `ToolbarSet()`, `ToolbarText()`, `ToolbarVisible()`

Toolbar()

This function displays the toolbar and waits for the user to click on a button. If button 0 has been defined with an associated function, that function is called repeatedly while no button is pressed. If no buttons are defined or enabled, or if all buttons become undefined or disabled, the toolbar is in an illegal state and an error is returned. If the toolbar was not visible, it becomes visible when this command is given.

If the user presses the "escape" key (Esc) with the toolbar active, the script will stop unless an "escape button" has been set by `ToolbarSet()`, in which case the action associated with that button is performed.

```
Func Toolbar(text$, allow% {,help%|help$});
```

`text$` A message to display in the message area of the toolbar. The area available for messages competes with the area for buttons. If there are too many buttons, the message may not be visible.

`allow%` A code that defines what the user can do (apart from pressing toolbar buttons). The code is the sum of possible activities:

1	User may swap to other applications.
2	User may change the current window.
4	User may move and resize windows.
8	User may use the File menu.
16	User may use the Edit menu.
32	User may use the View menu.
64	User may use the Analysis menu.
128	User may use the Cursor menu and add cursors.
256	User may use the Window menu.
512	User may use the Sample menu.
1024	User may not double-click y axis.
2048	User may not double-click the x axis or scroll it.
4096	User may not change channel of horizontal cursors.
8192	User may not change to another frame.

A value of 0 would restrict the user to the current view in a fixed window and, in a data view, the user would be able to scroll data and switch frames. A value of 9216 is the same as 0 but without being able to change y axes or frame.

`help` This is either a numeric code or a string that defines the help to be presented if the user asks for it while using the toolbar. A code of 0 means use standard help.

Returns The function returns the number of the button that was pressed to leave the toolbar, or a negative code returned by an associated function.

The buttons are displayed in order of their item number. Undefined items leave a gap between the buttons. This effect can be used to group related buttons together.

See also: `Interact(), ToolbarClear(), ToolbarEnable(), ToolbarSet(), ToolbarText(), ToolbarVisible()`

ToolbarClear()

This function is used to remove some or all of the buttons from the toolbar. If you delete all the buttons, the `Toolbar()` function will insert a button labelled OK so you can get out of the `Toolbar()` function. Use `ToolbarText("")` to clear the toolbar message.

```
Proc ToolbarClear({item%});
```

`item%` If present, this is the number of the button in the toolbar to clear. Buttons are numbered starting at 0. If omitted, all buttons in the toolbar are cleared.

See also: `Interact(), Toolbar(), ToolbarEnable(), ToolbarSet(), ToolbarText(), ToolbarVisible()`

ToolbarEnable()

This function enables and disables toolbar buttons and reports on the state of a button. Enabling an undefined button has no effect. If you disable all the buttons and then use the `Toolbar()` function or if you disable all the buttons in a function linked to the toolbar and there is no idle function set, a single OK button is displayed.

```
Func ToolbarEnable(item% {,state%});
```

`item%` The number of the button or -1 for all buttons. You must enable and disable button 0 with `ToolbarSet()` and `ToolbarClear()`.

`state%` If present this sets the button state. A value of 0 disables a button, a value of 1 enables a button.

Returns The function returns the state of the button prior to the call, as 0 for disabled and 1 for enabled. If all buttons were selected the function returns 0. If an undefined button, or button 0 is selected, the function returns -1.

See also: `Interact(), Toolbar(), ToolbarClear(), ToolbarSet(), ToolbarText(), ToolbarVisible()`

ToolbarSet()

This function adds a button to the toolbar and optionally associates a function with it. When a button is added, it is added in the enabled state.

```
Proc ToolbarSet(item%, label$ {,func ff%()});
```

`item%` The button number to add or replace. Buttons are numbered from 1-17. You can use an item number of 0 to set or clear a function that is called repeatedly while the toolbar waits for a button press. If you leave out a button this creates a gap between the adjacent buttons.

You can set an “escape” key as described in `Toolbar()`, by negating `item%`. For example `ToolbarSet(-2,"Quit");` sets button 2 as the escape key.

`label$` The button label plus optional key code and tooltip as `"Label|code|tip"`. Labels compete for space with each other; use tooltips for lengthy explanations. The label is ignored for button 0. Tooltips can be up to 79 characters long. To use a tooltip with no code use `"Label||A tooltip with no code field"`.

You can link a key to a button by placing `&` before a character in the label, or by adding a vertical bar and a key code in hexadecimal (e.g. 0x30), octal (e.g. 060) or decimal (e.g. 48) to the end of the label. Characters set by `&` are case insensitive. For example `"a&Maze"` generates the label aMaze and responds to `m` or `M`; the string `"F1:Go|0x70"` generates the label F1:Go and responds to the F1 key. Useful key codes include (nk = numeric keypad):

0x08 Backspace	0x09 Tab	0x0d Enter	0x1b Escape
0x20 Spacebar	0x21 Page up	0x22 Page down	0x23 End
0x24 Home	0x25 Left arrow	0x26 Up arrow	0x27 Right arrow
0x28 Down arrow	0x2e Del	0x30-0x39 0-9	0x41-0x5a A-Z
0x60-0x69 nk 0-9	0x6a nk *	0x6b nk +	0x6c nk separator
0x6d nk -	0x62 nk .	0x6f nk /	0x70-0x87 F1-F24

Use of other key codes or use of `&` before characters other than a-z, A-Z or 0-9 may cause unpredictable and undesirable effects.

Beware: When the toolbar is active, it owns all keys linked to it. If `A` is linked, you cannot type `a` or `A` into a text window with the toolbar active.

`ff%()` This is the name of a function. This function should have no arguments and the name with no brackets is given, for example `ToolbarSet(1,"Go",DoIt%);` where `func DoIt%()` is defined somewhere in the script. When the `Toolbar()` function is used and the user clicks on the button, the linked function is run (or if the `item%` 0 function is set, the function runs while no button is pressed.) The function return value controls the action of `Toolbar()` after a button is pressed.

If it returns 0, the `Toolbar()` function returns to the caller, passing back the button number. If it returns a negative number, the `Toolbar()` call returns the negative number. If it returns a number greater than 0, the `Toolbar()` function does not return, but waits for the next button. An item 0 function must return a value greater than 0, otherwise `Toolbar()` will return immediately.

If this argument is omitted, there is no function linked to the button. When the user clicks on the button, the `Toolbar()` function returns the button number.

See also: `Interact()`, `Toolbar()`, `ToolbarClear()`, `ToolbarEnable()`, `ToolbarText()`, `ToolbarVisible()`

ToolbarText()

This function replaces any message in the toolbar, and makes the toolbar visible if it is invisible. This function can be used to give a progress report on the state of a script that takes a while to run.

```
Proc ToolbarText(msg$);
```

`msg$` A string to be displayed in the message area of the toolbar.

See also: `Interact(), Toolbar(), ToolbarClear(), ToolbarEnable(), ToolbarSet(), ToolbarVisible()`

ToolbarVisible()

This function reports on the visibility of the toolbar, and can also show and hide it. You cannot hide the toolbar if the `Toolbar()` function is in use.

```
Func ToolbarVisible({show%});
```

`show%` If present and non-zero, the toolbar is made visible. If zero and the `Toolbar()` function is not active, the toolbar is made invisible.

Returns The state of the toolbar at the time of the call. The state is returned as 2 if the toolbar is active, 1 if it is visible but inactive and 0 if it is invisible.

See also: `Interact(), Toolbar(), ToolbarClear(), ToolbarEnable(), ToolbarSet(), ToolbarText()`

Trunc()

Removes the fractional part of a real number or truncates an array. To truncate a real number and return an integer value, just assign the real to the integer. To copy a real array to an integer array, use `ArrConst()`.

```
Func Trunc(x|x[]{[]...});
```

`x` A real number or a real array.

Returns When the argument is an array, the function replaces the array with the `Trunc` of all the points and returns either a negative error code or 0 if all was well.
When the argument is not an array the function returns the value with the fractional part removed. `Trunc(4.7)` is `4.0`; `Trunc(-4.7)` is `-4.0`.

See also: `Frac(), Round(), ArrConst()`

UCase$()

This function converts a string into upper case. The upper-case operation may be system dependent. Some systems may provide localised upper-casing, others may only provide the minimum translation of the ASCII characters a-z to A-Z.

```
Func UCase$(text$);
```

`text$` The string to convert.

Returns An upper-cased version of the original string.

See also: `LCase$()`

Val()

This converts a string to a number. The converter allows the same number format as the script compiler and leading white space is ignored.

```
Func Val(text${, &nCh%});
```

`text$` The string to convert to a number. The expected format is:
`{white space}{-}{digits}{.digits}{e|E{+|-}digits}`

`nCh%` If present, it is set to the number of characters used to construct the number.

Returns It returns the extracted number, or zero if no number was present.

See also: `Str$()`, `ReadStr()`

View(), View(v,c).[ ] and View().x()

The `View()` function sets the current view and returns the last view handle, or a negative error. A view handle is a positive integer > 0. Changing the current view does not change the focus or bring the view to the front; use `FrontView()` to do that.

```
Func View({vh%});
```

`vh%` An integer argument being:

- >0 A valid view handle that is to be made the current view. An invalid view handle will stop the script with a fatal error.
- 0 (Or omitted) no change of the current view is required.
- <0 If the argument is `-n`, this selects the `n`th duplicate of the current data view. This is equivalent to `Dup(n)`. Use `ViewSource()` to get the data view from which a memory view is derived.

Returns 0 if there are no views at all, -1 if the duplicate requested does not exist, otherwise it returns the view handle of the view that was current.

View(v,c).[ ]

The `View(vh%,c).[]` construction accesses view data for channel `c`. The `[]` refers to the whole array unless it encloses an expression to define a range of array elements. For waveform channels, the array holds the waveform values as expected. Marker channels appear as an array holding the marker times, but this array is read-only and a script error will be caused by attempting to assign to it. Use the `MarkTime()` function to change the times of markers.

```
View(vh%, c%).[{aExp}]
```

`vh%` A view handle of an existing view, 0 for the current view, or -n for the n^{th} duplicate view associated with the current view.

`c%` A channel number from the view.

`aExp` An optional array indexing expression. If omitted, the whole array is accessed.

Here are three examples, to work on data from bin `b%` in channel `c%` of view `v%`:

```
val:=View(v%,c%).[b%]             'get one data value
sum:=ArrSum(View(v%,c%).[b%:100])'sum 100 data values
ArrDiff(View(v%,c%).[])           'replace data by differences
```

View().x()

The `View().x()` construction overrides the current view for the evaluation of the function that follows the dot. It is an error if the selected view does not exist, and the script stops. Don't use this contruct for functions which close the view.

```
View(vh%).x()
```

`vh%` A view handle of an existing view, 0 for the current view (a waste of time), or -n for the n^{th} duplicate view associated with the current view.

`x()` A function or procedure.

For example, `View(vh%).Draw()` draws the view indicated by `vh%`. The equivalent code to `View(vh%).x()` is:

```
var temp%;
temp% := View(vh%);
x();
View(temp%);
```

See also: `FrontView()`, `ViewFind()`, `ViewKind()`, `ViewSource()`, `Window()`, `WindowTitle$()`

ViewColour()

This function gets and sets the colours of file, memory and XY view items, overriding the application-wide colours set by `Colour()`. Currently you can set the background colour.

```
Func ViewColour(item%{, col%});
```

`item%` The colour item to get or set; 0=background

`col%` If present, the new colour index for the item. There are 40 colours in the palette with indices 0 to 39. Use -1 to revert to the application colour for the item.

Returns The palette colour index at the time of the call or -1 if no view colour is set.

See also: `ChanColour()`, `Colour()`, `PaletteGet()`, `PaletteSet()`, `XYColour()`

ViewFind()

This function searches for a window with a given title and returns its view handle.

```
Func ViewFind(title$);
```

`title$` A string holding the view title to search for. Note that, in Windows 95 or 98, system settings in Windows Explorer can cause the file extension to be removed from the view title, so you may have to search both with and without the "`.cfs`".

Returns The view handle of a view with a title that matches the string, or 0 if no view matches the title.

See also: `FileOpen()`, `Window()`, `WindowTitle$()`, `View()`

ViewKind()

This function returns the type of a view or of the current view. Types 5-7 are reserved.

```
Func ViewKind({vh%});
```

`vh%` The handle of the view for which the type is required. If omitted the function returns information about the current view.

Returns The type of the view. View types are:

0	File view	A data view showing one frame at a time from a CFS data file. Access to any frame in the file is possible for analysis etc.
1	Text view	A view holding a text file for editing.
2	Output sequence	A view holding an output sequence file.
3	Script view	A view holding a script file.
4	Memory view	A data view created by a `SetXXX()` command, similar to a file view but wholly held in memory.
8	External text file	An invisible view for use with routines `Read()` and `Print()`.
9	External binary file	An invisible view for use by `BRead()`, `BWrite()` etc.
10	Application window	The Signal program window.
11	Other types	Other views with handles, such as the Status bar, and Toolbar, which can be made visible or invisible.
12	XY view	An XY view showing sets of XY data points.
-1	Unknown type	These include the cursor windows.
-2	Invalid handle	The return value if the `vh%` parameter value is invalid.

See also: `App(), ChanKind(), FileOpen(), View(), ViewList()`

ViewList()

This function fills an integer array with a list of view handles. It never returns the view handle of the running script. Use the `App()` command to get this.

```
Func ViewList({list%[] {,types%}});
```

`list%` An integer array that is returned holding view handles. The first element of the array, `list%[0]`, is set to the number of handles returned, and the remaining elements in the array are view handles. If the array is too small to hold the full number, the number that will fit are returned.

`types%` The types of view to include. This is a code that can be used to filter the view handles. The filter is formed by adding the types from the list below. If this is omitted or if no types are specified for inclusion, all view handles are returned.

1	File views	8	Script views	512	External binary	4096	XY views
2	Text views	16	Memory views	1024	Application view		
4	Sequencer	256	External text	2048	Other view types		

You can also exclude views otherwise included by adding:

8192	Exclude views not directly related to the current view.
16384	Exclude visible windows.
32768	Exclude hidden windows.
65536	Exclude duplicates.

Returns The total number of windows that satisfy the `types%`. This can be used to find a suitable array size.

See also: `App(), SampleHandle(), ViewKind()`

ViewSource()

This function returns the handle of the source view for a memory view.

```
Func ViewSource();
```

Returns The handle of the data view from which this memory view was derived, and from which it obtains its data. Note that memory views created using `SetCopy()` or `SetMemory()` do not have a data view attached and return 0.

See also: `View(), SetXXX(), SetAverage(), SetCopy(), SetMemory(), SetPower()`

ViewStandard()

This returns the current data or XY view to a standard state, by making all channels and axes visible in their standard drawing mode, axis range and colour. In an XY view the key is hidden and the axes are optimised. For other window types the result is undefined. If you use this without a current window, nothing happens.

```
Proc ViewStandard();
```

See also: `ChanOrder(), ChanWeight(), DrawMode(), XYDrawMode()`

ViewUseColour()

This function can be used to force the display to use black and white only, or to use the colours set in the colour dialog or by the `Colour()` command.

```
Func ViewUseColour({use%});
```

`use%` If present, a value of 0 forces Signal to display all windows in black and white. Any other value allows the use of colour. If omitted, no change is made.

Returns The current state as 1 if colour is in use, 0 if black and white is used.

See also: `Colour()`

Window()

This sets the position and size of the current view. Normally, positions are percentages from the top left-hand corner of the application window size. You can also set positions relative to a monitor. This can also be used to position, dock and float dockable toolbars.

```
Proc Window(xLow, yLow{, xHigh{, yHigh{, scr%{, rel%}}}});
```

`xLow` Position of the left hand edge of the window. When docking a dockable toolbar, the `xLow` and `xHigh` values correspond to the position of the top left corner of the window when dropped with the mouse.

`yLow` Position of the top edge of the window.

`xHigh` If present, the right hand edge. If omitted the previous width is maintained. If the window is made too small, the minimum allowed size is used. If the current view is a dockable control and `yHigh` is 0, values less than 1 or greater than 4 float the window at `(xLow, yLow)`, otherwise `xHigh` sets the docking state:

1	Docked to the left window edge	3	Docked to the right edge
2	Docked to the top window edge	4	Docked to the bottom edge

`yHigh` If present, the bottom edge position. If omitted the previous height is maintained. If the window is made too small, the minimum allowed size is used.

If the Window is dockable and `yHigh` is 0, this command sets the docked state of the window (see `xHigh.`) Otherwise the window is floated with the nearest allowed width that is no more than `xHigh-xLow`. If `xHigh-xLow` is 0 or negative `YHigh` sets the height of the dockable window.

`scr%` Optional screen selector for views, dialogs and the application window. If omitted, positions are relative to the application window. Otherwise, 0 selects the entire desktop rectangle and greater values select a particular screen rectangle (see `System()` for screen information.) If `rel%` is 1, positions are relative to the selected rectangle. If `rel%` is 0 or omitted, positions are relative to the intersection of the application window and this rectangle.

`rel%` Omit or set 0 for application window relative, 1 for screen or desktop relative.

This command can also position the application-window. If `scr%` is omitted, positions are relative to the primary monitor screen. If `scr%` is 0, positions are relative to the entire desktop, otherwise to screen `scr%`.

```
view(App()).Window(0,0,100,100);      'set maximum application size
```

See also: `App(), System(), WindowDuplicate(), WindowGetPos(), WindowSize(), WindowTitle$(), WindowVisible()`

WindowDuplicate()

This duplicates the current data view, creating a new window that has all the settings of the current view. It does not duplicate channels; these are shared with the existing window. The new window becomes the current view and is created invisibly.

```
Func WindowDuplicate();
```

Returns This command returns the view handle of the new window or a negative error code or 0 if no free duplicates (there is a limit of 9 duplicates per data view.)

See also: `Dup(), Window(), WindowGetPos(), WindowSize(), WindowTitle$(), WindowVisible(), View()`

WindowGetPos()

This gets the position of the current view. Positions are given as percentages of the available area, measured from the top left hand corner. Positions for the Signal application window are measured with respect to the whole screen; otherwise the positions are measured with respect to the Signal application area.

```
Proc WindowGetPos(&xLow, &yLow ,&xHigh ,&yHigh)
```

`xLow` A real variable that is set to the position of the left hand edge of the window.

`yLow` A real variable that is set to the position of the top edge of the window.

`xHigh` A real variable that is set to the position of the right hand edge of the window.

`yHigh` A real variable that is set to the position of the bottom edge of the window.

See also: `App(), Window(), WindowDuplicate(),WindowSize(), WindowTitle$(), WindowVisible()`

WindowSize()

This procedure is used to resize the current window without changing the position of the top left-hand corner. Setting a window dimension less than zero leaves the dimension unchanged. Setting a dimension smaller than the minimum allowed sets the minimum value. Setting a size greater than the maximum allowed sets the maximum size. There are no errors from this function. When this is used to size Signal application window the available area is the whole screen, otherwise the available area is Signal application area.

```
Proc WindowSize(width, height);
```

`width` The width of the window as a percentage of the available area.

`height` The height of the window as a percentage of the available area.

See also: `App(), Window(), WindowDuplicate(), WindowGetPos(), WindowTitle$(), WindowVisible()`

WindowTitle$()

This function gets and sets the title of the current window. There may be windows that are resistant to having their title changed. For these, the routine has no effect. Most windows can return a title. If you change a title, dependent window titles change, for example, cursor windows belonging to data views track the title of the data view.

```
Func WindowTitle$({new$});
```

`new$` If present, this sets the new window title. Window titles must follow any system rules for length or content. Illegal titles (for example titles containing control characters) are mangled or ignored at the discretion of the system.

Returns The window title as it was prior to this call.

See also: `Window(), WindowDuplicate(), WindowGetPos(), WindowSize(), WindowVisible()`

WindowVisible()

This function is used to get and set the visible state of the current view. This function can also be used on the application window, however the effect will vary with the system and on some, there may be no effect at all.

```
Func WindowVisible({code%});
```

`code%` If present, this sets the window state. The possible states are:

0 Hidden, the window becomes invisible. A hidden window can be sent data, sized and so on; the result is just not visible.

1 Normal, the window assumes its last normal size and position and is made visible if it was invisible or iconised.

2 Iconised. An iconised window can be sent data, sized and so on; the result is not visible.

3 Maximised. The window is made as large as possible.

4 Application window only; extend over all available desktop monitors.

Returns The window state prior to this call.

See also: `FrontView()`, `Window()`, `WindowDuplicate()`, `WindowGetPos()`, `WindowSize()`, `WindowTitle$()`

XAxis()

This function can be used to turn on and off the x axis of the current view, or to find the state of the x axis:

```
Func XAxis({on%});
```

`on%` Optional, set the axis state. If omitted, no change is made. Possible values are:

0 Hide the axis.
1 Show the axis with units in seconds.
2 Show with units of hh:mm:ss relative to start of file (not implemented yet).
3 Show with time of day (not implemented yet).

Returns The axis state at the time of the call (0 to 3, as above) or a negative error code. It is an error to use this function on a view that has no concept of an x axis.

Changes made by this function do not cause a redraw immediately. The affected view is drawn at the next opportunity.

See also: `XHigh()`, `XLow()`, `XRange()`, `XTitle$()`, `XUnits$()`

XAxisAttrib()

This function controls the options available through check boxes in the x-axis dialog for all views that have an x-axis. If you set values that would cause illegible or unintelligible axes, they are stored but not used unless other axis attributes change to make the values useful.

```
Func XAxisAttrib({flags%});
```

`flags%` A value of 0 sets a linear axis with no auto-adjust of units for high zoom. Add 1 for logarithmic. Add 2 to display powers on the logarithmic axis (you must have added 1 as well for this to take effect) Add 4 to cause a linear axis to auto-adjust its units at high zoom around 0. Omit this argument for no change to the attributes.

Returns The sum of the current flags set for the x-axis.

See also: `XAxis()`, `XAxisMode()`, `XAxisStyle()`, `XHigh()`, `XLow()`, `XRange()`, `YAxisAttrib()`

XAxisMode()

This function controls what is drawn in an x axis.

```
Func XAxisMode({mode%});
```

`mode%` Optional argument that controls how the axis is displayed. If omitted, no change is made. Possible values are the sum of the following. Values not included in the sum will be restored to their default states:

1 Hide all the title information.
2 Hide all the unit information.
4 Hide small ticks on the x axis. Small ticks are hidden if big ticks are hidden.
8 Hide numbers on the x axis. Numbers are hidden if big ticks are hidden.
16 Hide the big ticks and the horizontal line that joins them.
32 Scale bar axis. If selected, add 4 to remove the end caps.

Returns The x axis mode value at the time of the call or a negative error code.

See also: `XAxis(), XHigh(), XLow(), XRange(), YAxisMode()`

XAxisStyle()

This function controls the major and minor tick spacing for all views that have an x axis. If you set values that would cause illegible or unintelligible axes, they are stored but not used unless the axis range or scaling changes to make the values useful.

```
Func XAxisStyle({style%{, nTick%{, major}}});
```

`style%` A value of -1 returns the number of minor divisions set or 0 for automatic. A value of -2 returns the major tick spacing or 0 for automatic spacing. Set `style%` to 0 if setting `nTick%` or `major`.

`nTick%` The number of minor tick subdivisions or 0 for automatic spacing. Omit `nTick%` or set it to -1 for no change.

`major` If present, values greater than 0 set the major tick spacing. Values less than or equal to 0 select automatic major tick spacing.

Returns See the description of `style%` for return values.

See also: `XAxis(), XAxisAttrib(), XAxisMode(), XHigh(), XLow(), XRange(), YAxisStyle()`

XHigh()

This returns the value of the right end of the x axis of the current data or XY view. To find the frame limit use `Maxtime()`.

```
Func XHigh();
```

Returns The x axis upper limit in x axis units. It is a fatal error to use this function in an inappropriate view.

See also: `Draw(), XRange(), BinToX(), XToBin(), XLow(), Maxtime()`

XLow()

This returns the value that corresponds to the left end of the x axis of the current data or XY view. To find the frame limit use `Mintime()`.

```
Func XLow();
```

Returns The x axis lower limit in x axis units. It is a fatal error to use this function in an inappropriate view.

See also: `Draw(), XRange(), BinToX(), XToBin(), XHigh(), Mintime()`

XRange()

This function sets the start and end of the x axis in a data or XY view in x axis units. Unlike `Draw()`, it does not update the view immediately; updates must wait for the next `Draw()`, `DrawAll()` or some interactive activity.

```
Proc XRange(low {,high});
```

`low` The left hand edge of the view in x axis units.

`high` The right hand edge of the view. If omitted, the view stays the same width.

Values are limited to the axis range. Without `high`, it preserves the width, adjusting `low` if required. If the resulting width is less than the minimum allowed, no change is made.

See also: `Draw(), XLow(), XHigh()`

XScroller()

This function gets and optionally sets the visibility of the x axis scroller.

```
Func XScroller({show%});
```

`show%` If present, 0 hides the scroll bar and buttons, non-zero shows it.

Returns 0 if the scroll bar was hidden, 1 if it was visible.

XTitle$()

This function gets the title of the x axis. In a memory or XY view, or in a sampling document view, you can also set the title. The window will update with a new title at the next opportunity, but, in Version 3.00, the x axis title is not written to CFS data files.

```
Func XTitle$({new$});
```

`new$` If present this sets the new x axis title in a sampling document or memory view.

Returns The x axis title at the time of the call.

See also: `ChanTitle$(), XUnits$()`

XToBin()

This converts between x axis units and bin numbers in a data view.

```
Func XToBin(chan%, x);
```

`chan%` A channel number (1 to n).

`x` An x axis value. If it exceeds the x axis range it is limited to the nearer end.

Returns In a data view it returns the bin position that corresponds to x. In general, this will not be an integral number of bins; however, when used to access a bin, it will be truncated to an integer, and will refer to the bin that contains the x value.

See also: `BinToX(), BinSize(), BinZero()`

XUnits$()

This function gets the units of the x axis in the current view. In a memory, XY, or sampling document view, you can also set the units. The window will update with the new units at the next opportunity and they will become part of the new file if it is saved.

```
Func XUnits$({new$});
```

`new$` If present this sets the new x axis units in a new file or memory view.

Returns The x axis units at the time of the call.

See also: `ChanUnits$(), XTitle$()`

XYAddData()

This adds data points to an XY view channel. If the axes are set to automatic expanding mode by `XYDrawMode()`, they will change when you add a new data point that is out of the current axis range. If the channel is set to a fixed size (see `XYSize()`), adding new points causes older points to be deleted once the channel is full. The first form of the command allows unrestricted x and y positions. The second form is for data that is equally spaced in the x direction.

```
Func XYAddData(chan%, x|x[]|x%[], y|y[]|y%[]);
Func XYAddData(chan%, y[], xInc{, xOff});
```

`chan%` A channel number in the current XY view. The first channel is number 1.

`x` The x co-ordinate(s) of the added data point(s). In the first form of the command, both x and y must be either single variables or arrays. If they are arrays, the number of data points added is equal to the size of the smaller array.

`y` The y co-ordinate(s) of the added data point(s). In the second form of the command, this is an array of equally spaced data in x.

`xInc` Sets the x spacing between the y data points in the second form of the command.

`xOff` Sets the x position of the first data point in the second form of the command. If omitted, the first position is set to 0.

Returns The number of data points which have been added successfully.

See also: `XYColour(), XYCount(), XYDelete(), XYDrawMode(), XYGetData(), XYInCircle(), XYInRect(), XYJoin(), XYKey(), XYRange(), XYSetChan(), XYSize(), XYSort()`

XYColour()

This gets or sets the colour of a channel in the current XY view. The default channel colour is the same as for an ADC channel in a time window.

```
Func XYColour(chan% {,col%});
```

`chan%` A channel number in the current XY view.

`col%` The index of the colour in the colour palette. There are 40 colours in the palette; their indexes are numbered from 0 to 39. If omitted, there is no change of colour for the channel.

Returns The index of the colour in the colour palette or a negative error code.

See also: `Colour(), XYAddData(), XYCount(), XYDelete(), XYDrawMode(), XYGetData(), XYInCircle(), XYInRect(), XYJoin(), XYKey(), XYRange(), XYSetChan(), XYSize(), XYSort()`

XYCount()

This gets the number of data points in a channel in the current XY view. To find the maximum number of data points, see the `XYSize()` command.

```
Func XYCount(chan%);
```

`chan%` A channel number in the current XY view.

Returns The number of data points in the channel or a negative error code.

See also: `XYAddData(), XYColour(), XYDelete(), XYDrawMode(), XYGetData(), XYInCircle(), XYInRect(), XYJoin(), XYKey(), XYRange(), XYSetChan(), XYSize(), XYSort()`

XYDelete()

This command deletes a range of data points or all data points from one channel of the current XY view. Use `ChanDelete()` to delete the entire channel.

```
Func XYDelete(chan% {,first% {,last%}});
```

`chan%` A channel number in the current XY view.

`first%` The zero-based index of the first data point to delete. If omitted, all data points are deleted.

`last%` The zero-based index of the last data point to delete. If omitted, data points from `first%` to the last point in the channel are deleted. If `last%` is less than `first%` no data points are deleted.

Returns The function returns the number of deleted data points.

The index number of a data point depends on the current sorting method of the channel set by `XYSort()`. For different sorting methods, a data point may have different index numbers. The data points in a channel have continuous index numbers. When a point has been deleted the remaining points re-index themselves automatically.

See also: `ChanDelete(), XYAddData(), XYColour(), XYCount(), XYDrawMode(), XYGetData(), XYInCircle(), XYInRect(), XYJoin(), XYKey(), XYRange(), XYSetChan(), XYSize()`

XYDrawMode()

This gets and sets the drawing and automatic axis expansion modes of a channel in the current XY view. It is an error to use this command with any other view type.

```
Func XYDrawMode(chan%, which% {,new%});
```

`chan%` A channel number in the current XY view. This is ignored when `which%` = 5, as all the channels in an XY view share the same axes.

`which%` Which drawing parameter to get or set in the range 1-5. When setting parameters, the new value is held in the `new%` argument. The values are:

1 Get or set the data point draw mode. The drawing modes are:

0	dots (default)	1	boxes	2	plus signs +
3	crosses x	4	circles (NT only)	5	triangles
6	diamonds	7	horizontal line	8	vertical line

2 Get or set the size of the data points. The sizes allowed are 0 to 100 (0 is invisible.) The default size is 5.

3 Get or set the line style. If the line thickness is greater than 1 all lines are drawn as style 0. Styles are:

0 solid (default) 1 dotted 2 dashed

4 Get or set the line thickness. Thickness values range from 0 (invisible) to 10. The default is 1.

5 Get or set automatic axis range mode. This applies to the entire view, so the `chan%` argument is ignored. Values are:

0 The axes do not change automatically when new data points are added.

1 When new data points are added that lie outside the current x or y axis range, the data and axes screen area update at the next opportunity to display all the data.

`new%` New channel draw or axis expanding mode. If omitted, no change is made.

Returns The value of relevant channel draw mode or axis expanding mode at time of call if successful or a negative error code if it fails.

See also: `XYAddData(), XYColour(), XYCount(), XYDelete(), XYGetData(), XYInCircle(), XYInRect(), XYJoin(), XYKey(), XYRange(), XYSetChan(), XYSize(), XYSort()`

XYGetData()

This gets data points between two indices from a channel in the current XY view. It is an error to use this command with any other view type.

```
Func XYGetData(chan%, &x|x[], &y|y[] {,first% {,last%}});
```

`chan%` A channel number in the current XY view.

`x|x[]` The returned x co-ordinate(s) of data point(s). When arrays are used, either both `x` and `y` must be arrays or neither can be. The smaller of the two arrays sets the maximum number of data points that can be returned.

`y|y[]` The returned y co-ordinate(s) of data point(s).

`first%` The zero-based index of the first data point to return. If omitted, the first data point is index 0.

`last%` The zero-based index of the last data point returned. `last%` is only meaningful when `x` and `y` are array names. If omitted or if `last%` is greater than or equal to the number of data points, the final data point is the last one in the channel. If `last%` is less than `first%`, no data points are returned.

Returns The number of data points copied. This can be less than the number of data points between the requested indices. For example, if the size of `x` or `y` array is not big enough to hold all the data points from `first%` to `last%`, the number of data points returned is equal to the size of the array. If `x` and `y` are simple variables, 1 is returned if the data point with index number `first%` exists.

The index number of a data point depends on the current sorting method (see `XYSort()`.)

See also: `XYAddData(), XYColour(), XYCount(), XYDelete(), XYDrawMode(), XYInCircle(), XYInRect(), XYJoin(), XYRange(), XYKey(), XYSetChan(), XYSize(), XYSort()`

XYInCircle()

This gets the number of data points inside a circle defined by `xc`, `yc`, and `r` in the current XY view. A general point `(x,y)` is considered to be inside the circle if:

(x-xc)^2 + (y-yc)^2 <= r^2

Points lying on the circumference are considered inside, but owing to floating point rounding effects they may be indeterminate.

```
Func XYInCircle(chan%, xc, yc, r);
```

`chan%` A channel number in the current XY view.

`xc,yc` These are the x and y co-ordinates of the centre of the circle.

`r` This is the radius of the circle. `r` must be >= 0.

Returns The number of data points inside the circle or a negative error code.

See also: `XYAddData(), XYColour(), XYCount(), XYDrawMode(), XYGetData(), XYInRect(), XYJoin(), XYRange(), XYKey(), XYSetChan(), XYSize(), XYSort()`

XYInRect()

This function returns the number of data points in a channel of the current XY view that lie inside a rectangle. Data points on the boundaries of the rectangle are considered to be inside, but owing to floating point rounding they may be indeterminate.

```
Func XYInRect(chan%, xl, yl, xh, yh);
```

`chan%` A channel number in the current XY view.

`xl,xh` The `x` co-ordinates of the left and right hand edges of the rectangle. `xh` must be greater than or equal to `xl`.

`yl,yh` The `y` co-ordinates of the bottom and top edges of the rectangle. `yh` must be greater than or equal to `yl`.

Returns The number of data points inside the rectangle or a negative error code.

See also: `XYInCircle(), XYDelete(), XYDrawMode(), XYGetData(), XYJoin(), XYRange(), XYSetChan(), XYSize(), XYSort()`

XYJoin()

This function gets or sets the data-point joining method of a channel in the current XY view. Data points can be separated, joined by lines, or joined by lines with the last point connected to the first point (making a closed loop).

```
Func XYJoin(chan% {,join%});
```

`chan%` A channel number in the current XY view.

`join%` If present, this is the new joining method of the channel. If this is omitted, no change is made. The data point joining methods are:

0 Not joined by lines (this is the default joining method).
1 Joined by lines. The line style is set by `XYDrawMode()`.
2 Joined by lines and the last data point is connected to the first data point to form a closed loop.

Returns The joining method at time of call if successful or a negative error code if it fails.

See also: `XYAddData(), XYColour(), XYDrawMode(), XYRange(), XYKey(), XYSetChan(), XYSize(), XYSort()`

XYKey()

This gets and sets the display mode and positions of the channel key for the current view, which must be an XY view. The key displays channel titles (set by `ChanTitle$()`) and drawing symbols for all visible channels. It can be positioned anywhere within the data area. The key can be framed or unframed, transparent or opaque and visible or invisible.

```
Func XYKey(which%{, new});
```

`which%` This determines which property of the key we are interested in. Properties are :

1 Visibility of the key. 0 if the key is hidden (default), 1 if it is visible.
2 Background state. 0 for opaque (default), 1 for transparent.
3 Draw border. 0 for no border, 1 to draw a border (default).
4 Key left hand edge x position. It is measured from the left-hand edge of the x axis and is a percentage of the drawn x axis width in the range 0 to 100. The default value is 0.
5 Key top edge y position. It is measured from the top of the XY view as a percentage of the drawn y axis height in the range 0 to 100. The default is 0.

`new` If present it changes the selected property. If it is omitted, no change is made.

Returns The value selected by `which%` at the time of call, or a negative error code.

See also: `ChanTitle$(), XYAddData(), XYColour(), XYDrawMode(), XYJoin(), XYRange(), XYSetChan(), XYSize(), XYSort()`

XYRange()

This function gets the range of data values of a channel or channels in the current XY view. This is equivalent to the smallest rectangle that encloses the points.

```
Func XYRange(chan%, &xLow, &yLow, &xHigh, &yHigh);
```

`chan%` A channel number in the current XY view or -1 for all channels or -2 for all visible channels.

`xLow` A variable returned with the smallest x value found in the channel(s).

`yLow` A variable returned with the smallest y value found in the channel(s).

`xHigh` A variable returned with the biggest x value found in the channel(s).

`yHigh` A variable returned with the biggest y value found in the channel(s).

Returns 0 if there are no data points or the channel does not exist, 1 if values are found.

See also: `XYAddData()`, `XYColour()`, `XYCount()`, `XYDelete()`, `XYDrawMode()`, `XYGetData()`, `XYInCircle()`, `XYInRect()`, `XYKey()`, `XYSetChan()`, `XYSize()`, `XYSort()`

XYSetChan()

This function creates a new channel or modifies an existing channel in the current XY view. It is an error to use this function if the current view is not an XY view. This function can be used as a short-cut method for modifying all properties of an existing channel without calling the `XYSize()`, `XYSort()`, `XYJoin()` and `XYColour()` commands individually.

```
Func XYSetChan(chan% {,size% {,sort% {,join% {,col%}}}});
```

`chan%` A channel number in the current XY view. If `chan%` is 0, a new channel is created. Each XY view can have maximum of 256 channels, numbered from 1 upwards. The first channel is created automatically by Signal when you open a new XY view with `FileNew()`. If `chan%` is not 0, it must be the channel number of an existing channel which will be modified.

`size%` This sets the number of data points in the channel and how and if the number of data points can extend. The only limits on the number of data points are the available memory and the time taken to draw the view.

A value of zero (the default) sets no limit on the number of points and the size of the channel expands as required to hold data added to it.

If a negative size is given, for example `-n`, this limits the number of points in the channel to `n`. If more than `n` points are added, the oldest points added are deleted to make space for the extra points. If you set a negative size for an existing channel that is smaller than the points in the channel, points are deleted.

If a positive value is set, for example n, this allocates storage space for n data points, but the storage will grow as required to accommodate further points. Using a positive number rather than 0 can save time if you know in advance the likely number of data points, as it costs time to grow the data storage.

`sort%` This sets the sorting method of data points. The sorting method is only important if the points are joined. If they are not joined, it is much more efficient to leave them unsorted as sorting a large list of points takes time. The sort methods are:

0 Unsorted (default). Data is drawn and sorted in the order that it was added. The most recently added point has the highest sort index.

1 Sorted by x value. The index runs from points with the most negative x value to points with the most positive x value.

2 Sorted by y value. The index runs from points with the most negative y value to points with the most positive y value.

If this is omitted, the default value 0 is used for a new channel. For an existing channel, there is no change in sorting method.

`join%` If present, this is the new joining method of the channel. If this is omitted, no change is made to an existing channel and a new channel is given mode 0. The data-point joining methods are:

0 Not joined by lines (this is the default joining method).
1 Joined by lines. The line styles are set by `XYDrawMode()`.
2 Joined by lines and the last data-point is connected to the first data point to form a closed loop.

`col%` If present, this sets the index of the colour in the colour palette to use for this channel. There are 40 colours in the palette, their index numbered 0 to 39. If omitted, the colour of an existing channel is not changed. The default colour for a new channel is the colour that a user has chosen for an ADC channel in a time window.

Returns The highest channel number (including any created channel) that was affected by the command or a negative error code. When you create a new channel, the value returned is the number of the new channel.

See also: `XYAddData(), XYColour(), XYCount(), XYDelete(), XYDrawMode(), XYGetData(), XYInCircle(), XYInRect(), XYJoin(), XYKey(), XYRange(), XYSize(), XYSort()`

XYSize()

This function gets and sets the limits on the number of data points of a channel in the current XY view. Channels can be set to have a fixed size, or to expand as more data is added. The only limit on the number of data points is the available memory and the time taken to draw data.

```
Func XYSize(chan% {,size});
```

`chan%` A channel number in the current XY view.

`size%` This sets the number of data points in the channel and how and if the number of data points can extend. A value of zero sets no limit on the number of points and the size of the channel expands as required to hold data added to it.

If a negative size is given, for example `-n`, this limits the number of points in the channel to `n`. If more than `n` points are added, the oldest points added are deleted to make space for the extra points. If you set a negative size for an existing channel that is smaller than the points in the channel, points are deleted.

If a positive value is set, for example n, this allocates storage space for n data points, but the storage will grow as required to accommodate further points. Using a positive number rather than 0 can save time if you know in advance the likely number of data points, as it costs time to grow the data storage.

If this is omitted, there is no change to the size.

Returns If the number of points for the channel is fixed at n points, the function returns `-n`. Otherwise, the function returns the maximum number of points that could be stored in the channel without allocating additional storage space.

See also: `XYAddData(), XYColour(), XYCount(), XYDelete(), XYDrawMode(), XYGetData(), XYInCircle(), XYInRect(), XYJoin(), XYKey(), XYRange(), XYSetChan(), XYSort()`

XYSort()

In the current XY view, gets or sets the sorting method of the channel.

```
Func XYSort(chan% {,sort%});
```

`chan%` The channel number.

`sort%` This sets the sorting method of data points. The sorting method is only important if the points are joined. If they are not joined, it is much more efficient to leave them unsorted, as sorting a large list of points takes time. The sort methods are:

0 Unsorted (default). Data is drawn and sorted in the order that it was added. The most recently added point has the highest sort index.

1 Sorted by x value. The index runs from points with the most negative x value to points with the most positive x value.

2 Sorted by y value. The index runs from points with the most negative y value to points with the most positive y value.

If this is omitted, there is no change in sorting method.

Returns The function returns the sorting method at time of call or a negative error code.

See also: `XYAddData(), XYColour(), XYCount(), XYDelete(), XYDrawMode(), XYGetData(), XYInCircle(), XYInRect(), XYJoin(), XYKey(), XYRange(), XYSetChan(), XYSize()`

YAxis()

This function is used to turn the y axes on and off in the current view and to find the state of the y axes in a view.

```
Func YAxis({on%});
```

`on%` Optional argument that sets the state of the axes. If omitted, no change is made. Possible values are:

0 Hide all y axes in the view.
1 Show all y axes in the view.

Returns The state of the y axes at the time of the call (0 or 1) or a negative error code.

See also: `ChanTitle$(), ChanUnits$(), YHigh(), YLow(), YRange(), Optimise()`

YAxisAttrib()

This function controls the options available through check boxes in the y-axis dialog for all views that have y-axes. If you set values that would cause illegible or unintelligible axes, they are stored but not used unless other axis attributes change to make the values useful.

```
Func YAxisAttrib(cSpc(, flags%});
```

`cSpc` A channel specifier or -1 for all, -2 for visible and -3 for selected channels. When multiple channels are specified, returned values are for the first channel.

`flags%` A value of 0 sets a linear axis with no auto-adjust of units for high zoom. Add 1 for logarithmic. Add 2 to display powers on the logarithmic axis (you must have added 1 as well for this to take effect). Add 4 to cause a linear axis to auto-adjust its units at high zoom around 0. Omit this argument for no change to the attributes.

Returns The sum of the current flags set for the y-axis of the first channel in the list.

See also: `YAxis(), YAxisMode(), YAxisStyle(), YHigh(), YLow(), YRange(), XAxisAttrib()`

YAxisLock()

This function locks and unlocks the axes of grouped channels and reports on the locked state of grouped channels. If you lock a group, the grouped channels keep their own axis ranges, but display using the axis of the first channel in the group. The `YRange()`, `YHigh()` and `YLow()` commands operate on the information stored with a channel. To change the displayed range of grouped and locked channels, you must use `YRange()` on the first channel in a group.

```
Func YAxisLock(chan%{, opt%{, vOffs}});
```

`chan%` A channel that is in the group that you wish to address.

`opt%` If present, values of 1 and 0 set and unset the locked state. A value of -1 returns the visual offset per channel for the group. If omitted, no change is made.

`vOffs` If present, this sets the y axis display offset to apply between channels in the group. The `n`th channel has a visual offset of `(n-1)*offs`.

Returns The current locked state of the group unless `opt%` was -1, when it returns the y axis visual offset per channel for the group.

See also: `ChanOrder(), YAxisMode(), YHigh(), YLow(), YRange()`

YAxisMode()

This function controls what is drawn in a y axis and where the y axis is placed with respect to the data.

```
Func YAxisMode({mode%});
```

`mode%` Optional argument that controls how the axis is displayed. If omitted, no change is made. Possible values are the sum of the following values. Values not included in the sum will be restored to their default states:

1	Hide all the title information.
2	Hide all the unit information.
4	Hide y axis small ticks. They are also hidden when big ticks are hidden.
8	Hide y axis numbers. They are also hidden when big ticks are hidden.
16	Hide the big ticks and the vertical line that joins them.
32	Scale bar axis. If selected, add 4 to remove the end caps.
4096	Place the y axis on the right of the data.

Returns The state of the y axis mode at the time of the call or a negative error code.

See also: `ChanNumbers(), YAxis(), YAxisStyle(), YHigh(), YLow(), YRange()`

YAxisStyle()

This function controls the y-axis major and minor tick spacing. If you set values that would cause illegible or unintelligible axes, they are stored but not used unless the axis range or scaling changes to make the values useful.

```
Func YAxisStyle(cSpc, opt%{, major});
```

`cSpc` A channel specifier or -1 for all, -2 for visible and -3 for selected channels. When multiple channels are specified, returned values are for the first channel.

`opt%` Values greater than 0 set the number of subdivisions between major ticks. 0 sets automatic small tick calculation. Use -1 for no change. Values less than -1 return information, but do not change the axis style.

`major` If present and `opt%` is greater than -2, values greater than 0 sets the major tick spacing. Values less than or equal to 0 select automatic major tick spacing.

Returns If `opt%` is -2 this returns the current number of forced subdivisions or 0 if they are not forced. If `opt%` is -3 this returns the current major tick spacing if forced or 0 if not forced. Otherwise the return value is 0 or a negative error code. If multiple channels are specified the return value is for the first channel in the list.

See also: `YAxis(), YAxisAttrib(), YAxisMode(), YHigh(), YLow(), YRange(), XAxisStyle()`

Yield()

This function suspends script operation for a user-defined time and allows the system to idle. During the idle time, invalid screen areas update, you can interact with the program and the system has the opportunity to do housekeeping. If your script runs for long periods without using `Interact()` or `Toolbar()`, adding an occasional `Yield()` can make it feel more responsive and stop the operating system marking Signal as "not responding".

```
Func Yield({wait{, allow%}});
```

`wait` An optional time period, in seconds, to wait. If omitted or set to 0, the program will give the system one idle cycle and continue to run. If set negative, there is no idle cycle but the `allow%` argument is applied.

`allow%` This defines what the user can do during the wait period. See `Interact()` for the allowed values. The `allow%` value is cancelled after the command unless `wait` is 0 or negative.

Returns The function returns 1. We may add more return codes in future versions.

See also: `Interact(), Seconds(), TimeDate(), Toolbar()`

YieldSystem()

To share the system CPU(s) among competing processes, the operating system allocates time slices of around 10 milliseconds based on process priorities and recent process activity. A process can surrender a time slice if it has nothing to do. A typical application spends most of its time waiting for user input, which appears as messages in the application message queue; it will surrender a time slice if the message queue is empty unless it has a task to work on. Signal normally surrenders time slices, but if you run a script, it runs for the full time slice unless it is in `Yield()`, `Interact()`, or you use `ToolBar()` or `DlgShow()` without an idle routine.

The `YieldSystem()` command causes Signal to surrender the current time slice and suspends the user interface and script thread for a user-defined time or until a new message arrives in the Signal input queue. It has no effect on sampling, which runs in a separate thread. Unlike `Yield()`, it does not allow Signal to idle.

```
Proc YieldSystem({wait});
```

`wait` The time to wait, in seconds, before resuming the thread. Values are rounded to the nearest millisecond. Values greater than 10 are treated as 10 seconds; values less than –10 are treated as –10 seconds.

For `wait` values greater than 0, the wait is ended by unserviced messages; keyboard and mouse activity, timers for screen updates and the like cause messages. If `wait` is 0 or omitted, the current time slice is surrendered, but if Signal is the highest priority task it will be re-scheduled immediately. Negative values suspend Signal for `-wait` seconds regardless of messages.

`YieldSystem()` with wait values greater than 0 returns immediately if there are messages in the input queue. Unless you allow Signal to idle, either with a `Yield()` call or with `Toolbar()` or `DlgShow()`, there will always be pending messages, so it will have no effect. If you have a script loop that causes 100% CPU usage, inserting:

```
Yield();YieldSystem(0.001);
```

Will give other processes a chance to run. Increasing the wait time up to 0.05 will further reduce the CPU usage. Larger values have little additional effect due to timer messages ending the wait early. To give as much system time as possible to other tasks without allowing Signal to idle, you can use:

```
YieldSystem(-0.001);
```

In this case, increasing the time to –10.0 will have an effect; Signal will feel completely unresponsive until the time period has elapsed.

See also:`Interact()`, `DlgShow()`, `Seconds()`, `Toolbar()`, `Yield()`

YHigh()

This function returns the current upper limit of the y axis in a data or XY view.

```
Func YHigh(chan%);
```

`chan%` A channel number (1 to n). The channel number is ignored for an XY view.

Returns The value at the appropriate end of the axis.

See also: `ChanTitle$()`, `ChanUnits$()`, `YLow()`, `YRange()`, `Optimise()`

YLow()

This function returns the current lower limit of the y axis in a data or XY view.

```
Func YLow(chan%);
```

`chan%` A channel number (1 to n). The channel number is ignored for an XY view.

Returns The value at the appropriate end of the axis.

See also: `ChanTitle$(), ChanUnits$(), YHigh(), YRange(), Optimise()`

YRange()

This sets the y axis range for a channel or XY view. Attempting to set the range for a display mode that doesn't have a y axis is not an error, but has no effect. If the y range changes, the display is invalidated, but waits for the next `Draw()`.

```
Proc YRange(chan%|chan%[]|chan${, low, high});
```

`chan%` A channel number (1 to n) or you can also use -1 for all channels, -2 for all visible channels, -3 for all selected channels, -4 for waveform channels or -5 for marker channels, -6 for selected waveform channels or visible if none selected, -7 for visible waveform channels or -8 for selected waveform channels. The channel number is ignored in an XY view as there is only one axis for all channels.

This can also be an integer array. If it is, the first array element holds the number of channels in the list. This is followed by a list of positive channel numbers.

`chan$` A string to specify channel numbers, such as "`1,3..8,9,11..16`".

`low` The value for the bottom of the y axis. If omitted, and the channel type has a known range, Signal sets `low` and `high` to suitable limits. For example, for a waveform channel the limits are those set by the 16-bit nature of the data.

`high` The value for the top of the y axis. If `low` and `high` are the same, or stupidly close, the range is not changed.

See also: `YHigh(), YLow(), ChanScale(), ChanOffset(), Optimise()`

Index

—D—

—E—

—F—

—G—

—H—

—I—

—J—

—K—

—L—

—M—

—N—

—O—

—P—

—T—