The Spike2
script language

Version 7

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

Trademarks and Tradenames used in this guide are acknowledged to be the Trademarks and Tradenames of their respective Companies and Corporations.

Table of Contents

1 Introduction

What is a script?

For many users, the interactive nature of Spike2 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 peak after the second stimulus pulse from now.
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, 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 Spike2 environment. You can create scripts by example, or type them in by hand.

Hello world

Traditionally, the first thing written in every computer language prints "Hello world" to the output device. Here is our version that does it twice! To run this, use the File menu New command to create a new, empty script window. Type the following:

```
Message("Hello world");
PrintLog("Hello world");
```

Click the 📑 Run button to check and run your first script. If you have made any typing errors, Spike2 will tell you and you must correct them before the script will run. The first line displays "Hello world" in a box and you must click on OK to close it. The second line writes the text to the Log view. Open the Log view to see the result (if the Log window is hidden you should use the Window menu Show command to make it visible).

So, how does this work? Spike2 recognises names followed by round brackets as a request to perform an operation (called a *function* in computer-speak). Spike2 has around 400 built-in functions, and you can add more with the script language. You pass the function extra information inside the round brackets. The additional information passed to a function is called the function *arguments*.

Spike2 interprets the first line as a request to use the function Message() with the argument "Hello world". The message is enclosed in double quotation marks to flag that it is to be interpreted as text, and not as a function name or a variable name.

An argument containing text is called a *string*. A string is one of the three basic data types in Spike2. The other two are *integer* numbers (like 2, -6 and 0) and real numbers (like 3.14159, -27.6 and 3.0e+8). These data types can be stored in *variables*.

Spike2 runs your script in much the same way as you would read it. Operations are performed in reading order (left to right, top to bottom). There are also special commands you can insert in the script to make the it run round loops or do one operation rather than another. These are described in the script language syntax chapter.

Spike2 can give you a lot of help when writing a script. Move the text caret to the middle of the word Message and press the F1 key. Help for the Message() command appears, and at the bottom of the help entry you will find a list of related commands that you might also find useful.

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 Spike2 that the script language can manipulate. The running script is hidden from most commands, however you can obtain its handle using `App(3)` so you can show and hide it.

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 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.

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 Spike2 can monitor your actions and write the equivalent script. This is 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 Spike2 shows you the correct function to use for each activity.

For example, let us suppose you use the Script menu Turn Recording On option, open a data file, select interval histogram analysis mode of channel 3, process all data in the file and end with the Script menu Stop recording command. Spike2 opens a new window holding the equivalent script (we have tidied the output up a little and added comments):

```
var v6%,v7%;        'declare two integer variables (% means integer)
v6% := FileOpen("demo.smr", 0, 3);    'Open file, save view handle
Window(10, 10, 80, 50);               'Set a window position
v7% := SetInth(3, 100, 0.01, 0);      'Create invisible INTH view
WindowVisible(1);                     'make the INTH visible
Process(0, View(-1).Maxtime(), 0, 1); 'Add data to INTH view
```

Some of this is fairly straightforward. You can find the `FileOpen()`, `SetInth()`, and `Process()` functions described in this manual and they seem to map onto the actions that you performed. However, there is extra scaffolding holding up the structure.

In the first line, the `var` keyword creates two integer variables, `v6%` and `v7%`. These variables hold view handles returned by `FileOpen()` and `SetInth()`. Spike2 generates the names from the internal view numbers (so your script may not be exactly the same). The result of the functions is copied to the variable with the `:=` *operator*. In English, the second line of the script could be read as: *Copy the result of the FileOpen command on file demo.smr to variable v6%*.

The `SetInth()` command makes a new window to hold an interval histogram of the data in channel 3 with 100 bins of 0.01 seconds width and with the first bin starting at an interval of 0 seconds. The `WindowVisible(1)` command is present because the new window created by `SetInth()` is hidden. Spike2 creates invisible windows so that you can size and position them before display to prevent excessive screen repainting.

The `View(-1).` syntax accesses data belonging to a view other than the current view. The current view when the `Process()` command is used is the result view and we want to access the maximum time in the original time view. The negative argument tells the script system that we want to change the current view to the time view associated with this result view. The dot after the command means that the swap is temporary, only for the duration of the `Maxtime()` function.

Using recorded actions

You can now run this script with the control buttons. The script runs and generates a new result 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.

```
var fileH%, inthH%;                 'view handle variables
fileH% := FileOpen("", 0, 1);       'blank for dialog, single window
while fileH% > 0 do                 'FileOpen returns -ve if no file
   inthH% := SetInth(3, 100, .01, 0);WindowVisible(1);
   Process(0, View(-1).Maxtime(), 0, 1);
   Draw();                          'Update the INTH display
   fileH% := FileOpen("", 0, 1); 'ask for the next file
wend;
```

This time, Spike2 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 Spike2 to draw the data after it calculates the interval histogram. There is a problem with this script if you open a file that does not contain a channel 3 that holds events or markers of some kind. 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 histogram, so the next step is to delete it. We also have added a line to close down all the windows at the start, to reduce the clutter when the script starts.

```
var fileH%, inthH%;
FileClose(-1);                      'close all windows except script
fileH% := FileOpen("", 0, 1);       'use a blank name to open dialog
while fileH% > 0 do                 'FileOpen returns -ve if no file
   inthH% := SetInth(3, 100, .01, 0);WindowVisible(1);
   Process(0, View(-1).Maxtime(), 0, 1);
   View(fileH%).FileClose();        'Shut the old window
   Draw();                          'Update the INTH display
   fileH% := FileOpen("", 0, 1); 'ask for the next file
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 3, 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%, inthH%, 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 for INTH", 126); 'all but waveform
   if (DlgShow(chan%) > 0) and   'User pressed OK and...
      (chan% > 0) then            '...selected a channel?
      inthH% := SetInth(chan%, 100, .01, 0);WindowVisible(1);
      Process(0, View(-1).Maxtime(), 0, 1);
      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.

Differences between systems

If you need to write code that can run on different systems, you should try to avoid system dependent features, or if this is impossible, make use of the `System()` function to find out where you are.

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. It is also used when a function can accept an array with any number of dimensions:

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

This means that the function `Sin()` will accept a real value or and array with one or more dimensions.

Sources of script information

You will find that the rest of this manual is a reference to the script language and to the built-in script functions. Once you are familiar with the scripting system it will be your most useful documentation. There are example and utility script provided with Spike2. These are copied to the `scripts` folder within the folder that contains Spike2.

There is a separate manual provided with Spike2 that has been used for our Spike2 training courses, held at CED and around the world. This manual contains many annotated examples and tutorials. Some of the scripts in this manual are useful in their own right; others provide skeletons upon which you can build your own applications.

Our web site at `www.ced.co.uk` has example scripts and script updates that you can download.

2 The script window and debugging
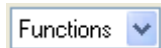
Script window

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. If a script is opened from a read only medium, or is write protected, you will not be allowed to edit it.

The script window is a text window with a few extra controls including a folding margin that allows you to fold away inner loops, function and procedures. The folding margin can be hidden; see the Edit menu Preferences for details of configuring the script folding margin.

Margin (breakpoints and bookmarks)

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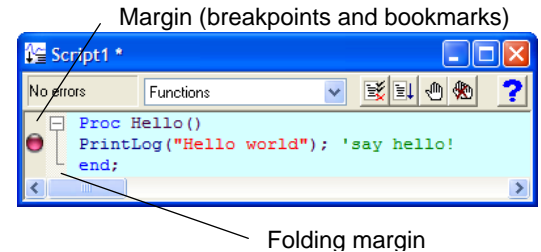
To the left of the text area is a margin where you can set break points (one is shown already set), bookmarks and where the current line of the script is indicated during debugging. Above the text area is a message bar and several controls. The controls have the following functions:

Function

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. Spike2 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, Spike2 runs the compiled version, starting from the beginning. Spike2 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 clicking the mouse pointer in the margin or by right clicking the line and using the *Toggle break* command.

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 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

Spike2 supports syntax colouring for both the script language and also for the output sequencer editor. You can customise the colouring (or disable it) from the *Script files* settings section of the Edit menu Preferences in the General tab. The language keywords have the standard colour blue, quoted text strings have the standard colour red, and comments have the standard colour green. You can also set the colour for normal text (standard colour black) and for the text background (standard colour white).

The syntax colouring options are saved in the Windows registry. If several users share the same computer, they can each have their own colouring preferences as long as they log on as different users.

Editing features for scripts

There are some extra editing features that can help you when writing scripts. These include automatic formatting, commenting and un-commenting of selected lines, code folding, auto-complete of typed words and pop-up help for built-in and user-defined functions. These are described in the documentation for the Edit menu.

Debug overview

Despite all our efforts to make writing a script easy, and all your efforts 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, Spike2 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
- Enter the debugger on a script error to view variable values
- View local and global variables
- Watch a selected set of 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

A Spike2 script 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 where you want to enter the debugger. Alternatively, you can enter the debugger "on demand" 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 Esc and click on a button. This is a very useful if your script is running round in a loop with no exit! You can disable this 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.

You can also choose to enter debug on a script error by checking the Enter debug on script error box in the General tab of the Edit menu Preferences option. Depending on the error, this may let you check the values of variables to help you fix the problem.

When your script enters the debugger, the debug toolbar opens if it was not visible. The picture shows the toolbar as a floating window, but you can dock it to any side of the Spike2 window by dragging over the window edge.
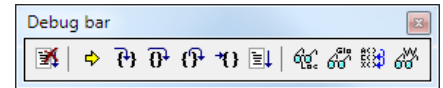
Stop running the script (`Ctrl+Alt+Shift+F5`). There is no check that you really meant to do this; we assume that if you know enough to open the debugger, you know what you are doing! You can use the `Debug()` command to disable the debugger.

Display the current script line (`Ctrl+Alt+F2`). If the script window is hidden, this makes it visible, brings it to the top and scrolls 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 (`Ctrl+F5`). 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 (`Shift+F5`). 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.

Step out of a procedure or function (`Alt+F5`). 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 to the start of the line with the text caret (`Ctrl+Alt+F2`). This is slightly quicker than setting a break point, running to it, then clearing it (which is how this is implemented). This is disabled unless the current view is a script file that is part of the currently running script.

Run the script (`Ctrl+Shift+F5`). This disables the buttons on the debug toolbar and the script runs until in reaches a break point or the end of the script.

Show the local variables for the current user-defined `func` or `proc` (`F7`). If there is no current routine, the window is empty. Double-click a variable to edit it. The first few hundred characters of array elements are displayed. If an array is longer than this, the display ends with … indicating more text. Right-click variables to add them to the Watch window.

Show the global variable values in a window (`Ctrl+F7`). 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. Right-click variables to add them to the Watch window.

Display the call stack (list of calls to user-defined functions with their arguments) on the way to the current line in a window (`Ctrl+Shift+F7`). If the `Toolbar()` function has been used, the function name is `<Callback>`.

Display the Watch window. Right-click variables in the Globals and Locals windows to add them to the Watch window. Right-click in the Watch window to control the display of watched variables.

The debug toolbar and the locals, globals and the call window close at the end of a script. Buttons are disabled if they cannot be used. Hover the mouse pointer over a button to display a Tool tip; if the Status bar is visible, a longer description can be seen there.

Enter debug on error   There is an option in the Edit menu Preferences dialog General tab to enter the debugger on an error. After an error you can inspect the values of local and global variables and the contents of the call stack, but you are not allowed to continue running the script.
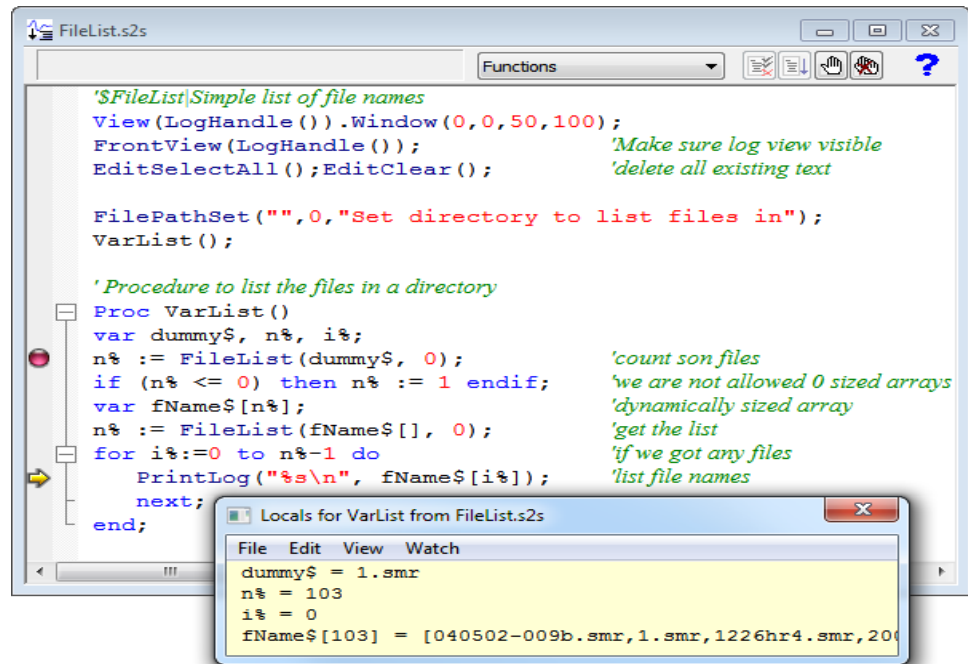
The example above shows a script that prompts the user to select a directory, opens it and prints a list of spike2 data files to the log window. The user set a break points, ran the script then clicked on the Step button four times and clicked the Locals button.

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 choose the element by editing the index fields, one for each dimension.

Watch window

Variables added to the Watch window are remembered between debug sessions. Variables that do not exist in the current script are marked. Right-click in the Watch window to see the options for managing watched variables.

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. This example calculates factorials using recursion. We have set a break point and then displayed

the call stack so you can see all the calls, and the arguments for each call. 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
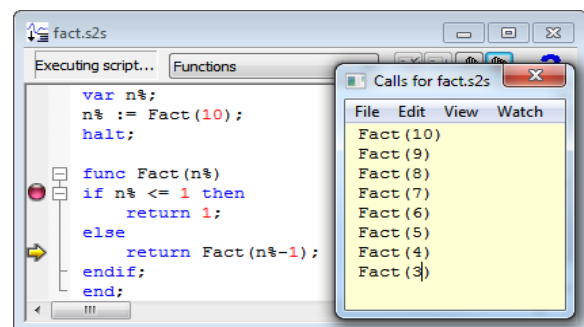
Script format A script consists of lines of text. Each line can be up to 240 characters long, however we suggest a maximum line length of 78 characters as experience shows that this makes printing and transfer of scripts to other systems simple.

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      break     case
const     continue  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 Spike2 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 cannot write `-2147483648` in a script as this is treated as minus `2147483648`, which overflows integer range. Write `-2147483647-1` or `0x8000000` instead.

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.

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 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 (but see `resize`, below). The dimensions of local arrays can be set by variables or calculated expressions. Simple variables (not arrays) can be initialised when they are declared. Before Spike2 version 7, the initialising expression could not include variables or function calls. Uninitialised numeric variables are 0; uninitialised strings are empty.

```
var Sam%:=3+2, pi:=4*ATan(1), sal$:="I am \"Sally\"";
```

Constant declarations

Constants are created by the `const` keyword. A constant can be of any of the three basic data types, must be initialised in the declaration with a constant expression and cannot be an array. A constant expression is composed of previously defined numeric constants, numbers and the operators add, subtract, multiply and divide (`+-*/`) or is a string constant. 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, pi:=3.1415926535897932, sal$:="I am \"Sally\"";
```

Arrays of data

The three basic types (integers, reals and strings) can be made into arrays with from 1 to 5 dimensions. Before Spike2 version 6, 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], text$[23];
```

declares a 10 element vector, a matrix with 10 rows and 1000 columns, a 5-dimensional array with 720 elements and an array of strings. 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; text$[1]:"a";
```

You can declare an array with one or more dimensions set to 0! However, such an array cannot be used in this state. All dimensions must have non-zero size before you can refer to an array in anything other than a `var` or `resize` statement. You can resize an array with the `resize` statement.

There is a maximum number of elements (product of the sizes of the dimensions) that you are allowed in an array. This is currently set to 100,000,000 in an attempt to prevent operations that would likely take a very long time.

The dimension sizes for an array declared outside a `Proc` or `Func` (a global array) must all be constant; inside a `Proc` or `Func` they can be variables. For example:

```
Proc VariableSizeArray(n%)
var x[n%];
```

You cannot have two `var` statements that refer to the same variable in the same context. That is, you cannot have code like:

```
var fred[23][32];
...
var fred[23[48];                'This line will generate an error
```

as this will generate a "Name multiply defined or redefined" error. In a `Proc` or `Func`, you can declare an array inside a loop, and change the size of the dimensions each time around the loop. However, version 7 provides the `resize` statement, and we urge you to declare arrays outside loops and use `resize` to do any required size changes.

```
Proc BadStyle()                 Proc BetterStyle()
var i%;                         var arr[0], i%;
for i% := 1 to 100 do           for i% := 1 to 100 do
   var arr[i%];                    resize arr[i%];
   ...                             ...
next;                           next;
end;                            end;
```

We may make resizing an array using `var` illegal in the future. Note that before Spike2 version 7, resizing with `var` preserved the original data when the last dimension was changed, but changes to any other dimension would not preserve the data.

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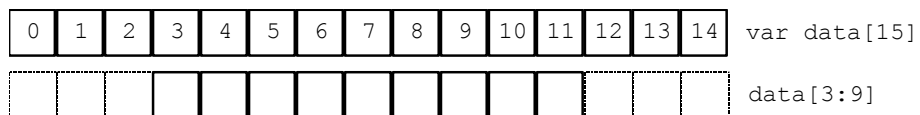
Matrix subsets With a matrix you have more options. You can pass a single element, a vector sub-set, or a matrix sub-set. Consider the matrix of real numbers defined as `var sq[6][7]`. You can pass this as a vector or a matrix to a function as (`a`, `b`, `c` and `d` are integer numbers):

`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]`.
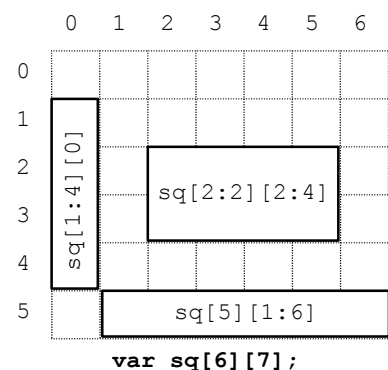
N-Dimensional array subsets

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 an array

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 Spike2 version 6 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 or array

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 6 of Spike2 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]`.

Resize array

You cannot change the number of dimensions of an array, but you can change the size of the dimensions. This is done with the `resize` statement (added at version Spike2 version 7), which has a syntax that is very similar to `var`:

```
resize v[24], M[2][3000], nd[6][5][4][3][2], text$[923];
```

When used in this way, the values in the square brackets, which can be expressions or constants, set the new size for each dimension. However, if you want to leave a dimension at the current size, you can use:

```
resize nd[][][][][n%];   'change last dimension only
```

A pair of empty square brackets means that you want to preserve the current size of the corresponding dimension. The `resize` statement preserves data in the array (unless you make one or more dimensions smaller, when data is omitted). When you make dimensions larger, new numeric array elements are set to 0; new string elements are set to an empty string (`""`).

In most cases, you will only want to change one dimension to cope with adding more items to an array. It is more efficient to increase the last dimension as in this case it is often possible to extend (or reduce) the memory allocated to the array without physically moving it in memory. If you change any other dimension than the last, the `resize`

statement allocates a new array of the required size, copies data into it, replaces the original array with the new one and releases the memory used by the original array.

You can always resize a global or local array unless a sub-array, transpose or diagonal of it has been passed to a `Proc` of `Func` and is currently in use. You will get the error message: "Attempt to index non-array or resize sub-array or result view" if you break this rule. Here are some examples to make this clearer:

```
var global[2][3];
Level1(global);            'pass entire global array
resize global[3][3];       'OK

proc Level1(g[][])         'g is entire global array
var local[3][4];
TryResize(local);          'this is OK, passing entire array
TryResize(g);              'this is OK, passing entire array
TryResize(global);         'this is OK, passing entire array
TryResize(local[:2][]);    'will fail as is a sub-array
ObscureError(local[:2][]); 'pass sub-array of local, OK
ObscureError(g[:1][:1]);   'pass sub-array of global, not OK
end;

Proc TryResize(arr[][])
resize arr[][2];
end;

Proc ObscureError(h[][])   'will fail in the resize...
resize global[4][];        '...if h is a sub-array of global
end;
```

When you create a sub-array, transpose or diagonal of an existing array, a temporary array construct is created that depends on the original. If you were to resize the original, all the dependant arrays that referred to it would become invalid, so we do not allow you to make such a change. You cannot resize an array derived from a result view.

Result views as arrays

The script language treats a result 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 result 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 or a `resize` statement
- 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
```

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 has 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 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 to concatenate strings (join them together).

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

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;` (which was allowed in the MS-DOS version of Spike2).

The ternary operator The ternary operator `?:` was added at Spike2 version 6 and has the following format:

numeric expression **?** *expression1* **:** *expression2*

The result of the ternary operator is *expression1* if *numeric expression* evaluates to a non-zero result otherwise it is *expression2*. You can use this anywhere that an expression would be acceptable, including when passing arrays as arguments to functions. However, *expression1* and *expression2* must be type compatible in the context of their use. For example, if one is a string, then the other must also be a string. If they are arrays passed as arguments, they must have the same type and the same number of dimensions. If they are arguments passed by reference, they must have identical type. In an expression, one can be integer and the other real, in which case the combined type is treated as real.

You are not allowed to use this operator to choose between function or procedure names passed as arguments into functions or procedures.

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 *expression* is non-zero, the statements after the then are executed. If *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 are equivalent
if num% mod 2 then num%:=num%+2 else num%:=num%+1 endif;
num% += num% mod 2 ? 2 : 1;'An alternative using ?:
```

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.

continue, break   Within the three looping statements (`for...next`, `repeat...until` and `while...do`) you can use the `break` and `continue` statements. The `break` statement jumps out of the enclosing looping statement; the `continue` statement jumps to the evaluation of the expression that determines if a `repeat` or `while` will run again and to the `next` in a `for` loop. For example:

```
var i%;
for i% := 0 to 1000 do
   if StopEarly() then break endif;
   if SkipToNext() then continue endif;
   DoSomething(i%);
next;                    'continue jumps here

DoSomethingElse();    'break jumps to this statement
```

It is an error to use either of these statements outside a loop. If you need to jump out of more than one level of looping statements, put the code in a `Proc` or `Func` and use the `return` statement to jump out. The `break` and `continue` statements are new in Spike2 version 7.

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$(1));        'Show the channel title
PrintLog(ChanComment$(1));      'and the comment
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%];       '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.

In a result view, channel numbers start at 1, but we accept 0 (meaning 1) to avoid breaking scripts written for older versions of Spike2. 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.

Include files

There are times when you will want to reuse definitions or user-defined procedures and functions in multiple projects. You can do this by pasting the text into your script, but it can be more convenient to use the `#include` command to include script files into a script. A file that is included can also include further files. We call these *nested* include files. Only the first `#include` of a file has any effect. Subsequent `#include` commands that refer to the same file are ignored. This prevents problems with script files that include each other and stops multiple definitions when two files include a common file. A `#include` command must be the first non-white space item on a line. There are two forms of the command:

```
#include "filename" 'optional comment
#include <filename> 'optional comment
```

where `filename` is either an absolute path name (starting with a `\` or `/` or containing a `:`), for example `C:\Scripts\MyInclude.s2s`, or is a relative path name, for example `include.s2s`. The difference between the two command forms lies in how relative path names are treated. The search order for the first form starts at item 1 in the following list. The search for the second form starts at item 3.

1. Search the folder where the file with the `#include` command lives. If this fails...
2. Search the folder of the file that included that file until we reach the top of the list of nested include files. If this fails...
3. Search any `Spike7\include` folder in the users document folder (`My Documents`) if it exists. If this fails…
4. Search any `\include` folder in the Spike7Shared folder inside the pblic documents folder. If this fails…
3. Search any `\include` folder in the folder in which Spike2 is installed. If this fails...
4. Search the current folder.

Included files are always read from disk, even if they are already open. If you have set the Edit menu Preferences option to save modified scripts before running, modified include files are automatically saved when you compile. If this option is not set, the script compiler will stop the compilation with an error if it finds a modified include file. You must save the included file to compile your script.

There are no restrictions on what can be in an included file. However, they will normally contain constant and variable definitions and user-defined procedures and functions. It is usually a good idea to have all your `#include` commands at the start of a script so that anyone reading the source is aware of the scope of the script.

The `#include` command was added to Spike2 at version 6.03 and is not recognised by any version before this. A typical file using `#include` might start with:

```
'$Example|Example of use if include files
#include <sysinc.s2s>          'my system specific includes
#include "include\proginc.s2s" 'search script relative folder
var myvar;                      'start of my code...
```

Opening included files If you right-click on a line that holds an include command, and Spike2 can locate the included file, the context menu will hold an entry to open the file. The search for the file follows that described above, except that it omits step 2.

Include files and debugging

You can debug a script that uses included files in exactly the same way as one that does not. If you step into a user-defined function or procedure that is in an included file, the included file will open and the stepping marker will show where you have reached. If you

want to set a break point in an included file before you run the script, open the included file, set the break point and then run the script (leaving the included file open).

Included files that are open within Spike2 are hidden from the script `ViewList()` command in the same way that the running script is not visible. However, unlike the running script, whose view handle can be obtained with the `App()` command, there is currently no mechanism built into the system to let you find the handles of included files.

Program size limits

The use of `#include` makes it much easier to construct very large scripts. However, there are some limits on the size of a script that you should bear in mind before trying to amalgamate every script you have ever written into one "superscript":

1. The maximum instructions in a compiled script is currently set at 1000000. The number of instructions your script uses is displayed when you compile a script.

2. A script file can contain up to 1000000 lines and you can have up to 4095 included files. However, you will almost certainly run out of instruction space before you hit these limits.

3. Every variable, constant, procedure and function you declare generates a script object. The maximum number of objects is currently around 65000.

4. The maximum number of characters in distinct literal strings is around 1000000. Before version 6.04 the limit was around 65000 characters.

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 open, close, manipulate, position, display, size, colour and create windows (views). These commands apply to windows in general. See the sections on Time, Result, XY and Text windows for more specific commands.

App	Get the application view handle and version and special views
ChanNumbers	Hide and show channel numbers
ColourGet/Set	Control basic system colours and the colour picker
Draw	Draw invalid regions of the view (and set x axis range)
DrawAll	Update all invalid regions in all views
EditCopy	Copy window to clipboard
FileClose	Closes a window or windows
FileComment$	Gets and sets the file comment for time views
FileConvert$	Convert a foreign file to a Spike2 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, time and result views
FileQuit	Closes the application
FileSave	Save a view
FileSaveAs	Save a view in variety of formats
FontGet	Read back information about the font
FontSet	Set the font for the current window
FocusHandle	Get handle of script-controllable window with the focus
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
Modified	Get and set the modified and read only state of views
PaletteGet	Get the RGB colour of a palette entry
PaletteSet	Set the RGB colour of a palette entry
View	Change or override current view and get view handle
ViewColourGet/Set	Override application colours for a view
ViewFind	Get a view handle from a view title
ViewKind	Get the type of the current view
ViewList	Form a list of handles to 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
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 visibility of the x axis
XAxisMode	Control how the x axis is drawn
XAxisStyle	Control x axis tick spacing and time view hh:mm:ss mode
XScroller	Get or set visibility of x axis scroll bar
XUnits$	Get units of the x axis
YAxis	Get or set the visibility of the y axis
YAxisMode	Control how the y axis is drawn
YAxisStyle	Control y axis tick spacing

Channel commands These commands manipulate data channels and get channel information.

BinError	Get or set error bar information
Binsize	The sample interval for waveforms, otherwise time resolution
BurstMake	Extract bursts to a memory channel from an event channel
BurstRevise	Modify a list of bursts in a memory channel
BurstStats	Collect statistics on bursts in preparation for BurstRevise
Chan$	Get the symbolic name for a channel
ChanCalibrate	Calibrate waveform or WaveMark channels from known values
ChanColourGet/Set	Override drawing mode based channel colours
ChanComment$	Get or set the channel comment
ChanData	Fill an array with a waveform or event times
ChanDelete	Delete a channel from a time view
ChanDuplicate	Duplicate channels in a time view
ChanHide	Make a channel invisible
ChanImage	Set a background image for a channel
ChanIndex	Select which RealMark item to display and measure
ChanKind	Get the type of a channel
ChanList	Get a list of channels meeting a specification
ChanMeasure	Make the same measurements as the Cursor Regions dialog
ChanNew	Create a new channel for ChanWriteWave() to write to
ChanOffset	Set waveform and WaveMark value for input of zero
ChanOrder	Modify the channel order and y axis grouping
ChanPenWidth	Set the pen width for a time or result view channel
ChanPixel	Get pixel size in channel units
ChanProcessAdd	Add new processing option to a waveform or RealWave channel
ChanProcessArg	Read or modify an argument of a channel process
ChanProcessClear	Delete a process, all processes for a channel, or all processes
ChanProcessCopy	Copy channel processes (rectify, smooth) to other channels
ChanProcessInfo	Get information about the processes attached to a channel
ChanPort	Get the physical sampling port of a time view data channel
ChanScale	Set waveform and WaveMark channel scale factor
ChanSave	Copy time view channels between views with optional time shift
ChanSearch	Search a channel for a feature (peak, level crossing, slope…)
ChanSelect	Select and report on selected state of channels
ChanShow	Make a channel visible
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
ChanWriteWave	Write data to a waveform or RealWave channel
Count	Count items in a time range, sum bins in result view
DrawMode	Get or set display mode for a channel
DupChan	Get information about duplicate channels
FitLine	Fit a straight line to waveform or result channel
LastTime	Find the previous item in a channel (and return values)
Maxtime	Time of last item on the channel
MemDeleteItem	Delete one or more items from a memory channel
MemDeleteTime	Delete one or more items based on time in a memory channel
MemGetItem	Get information on item in memory channel
MemImport	Import items into a memory channel
MemSave	Write the contents of a memory channel to the data file
MemSetItem	Edit or add an item in a memory channel
Minmax	Find minimum and maximum values (and positions)
NextTime	Find the next item in a channel (and return values)
VirtualChan	Create a channel from combinations of existing channels

Time views These commands manipulate time views.

BinToX	Convert time units to x axis units
DrawMode	Get or set display mode for a channel
Dup	Get the view handle of duplicates of the current view
EventToWaveform	Create a waveform channel from events
ExportChanFormat	Set channel format for export
ExportChanList	Set list of channels for export
ExportRectFormat	Set rectangular time view export
ExportTextFormat	Set format for text output of channels
FileApplyResource	Apply a resource file to the current time view
FileGlobalResource	Set a global resource file
FileSaveResource	Create a resource file matching the current time view
FileTimeBase	Get and optionally set the underlying time units
Maxtime	Maximum x axis value for any channel in the view
MarkEdit	Edit marker codes and other information
MarkInfo	Get information on extended marker types
MarkMask	Set the marker filter for a channel
MarkSet	Set the marker codes for data in a time range
MemChan	Create a channel in memory
Optimise	Set reasonable y range for channels with axes
ReRun	Get or set the rerunning state of a channel
ViewOverdraw	Add a list of trigger times to an overdrawn view
ViewOverdraw3D	Set the overdrawing parameters for overdrawn data in 3D mode
ViewTrigger	Control on-line triggered displays and off-line display stepping
VirtualChan	Create a channel from combinations of existing channels
WindowDuplicate	Duplicate a time view
XLow	Time of the start of the displayed area in seconds
XHigh	Time of the end of the displayed area in seconds
XRange	Set the x axis range for next draw
XScroller	Show or hide the x axis scroll bar and controls
XTitle$	Get x axis title
XToBin	Convert x axis units to time units
YAxisLock	Get and set the locking state of grouped channels
YRange	Set y axis range for a channel
YHigh	Get upper limit of y axis for a channel
YLow	Get lower limit of y axis for a channel

Result views These commands manipulate result views. Result views can also be treated as arrays; so all the array arithmetic commands are available.

BinError	Get or set error bar and bin count information
Binsize	Width of each bin in the x direction
BinToX	Convert bin number to x axis value
DrawMode	Get or set display mode for a channel
Maxtime	Number of bins in the result view
Minmax	Find minimum and maximum values (and positions)
Optimise	Set reasonable y range for display
RasterAux	Deprecated, use RasterSort() and RasterSymbol()
RasterGet	Read back raster information
RasterSet	Set raster information
RasterSort	Get and set the sorting values
RasterSymbol	Get and set the time values for raster symbols
Sweeps	Gets and sets the number of items added to result view
XHigh	Bin number of the end of the displayed area
XLow	Bin number of the start of the displayed area
XRange	Set the x axis range for next draw

XScroller	Show or hide the x axis scroll bar and controls
XTitle$	Get and set x axis title
XToBin	Convert x axis value to bin number
YAxisLock	Get and set the locking state of grouped channels
YHigh	Get upper limit of y axis
YLow	Get lower limit of y axis
YRange	Set y axis range

XY views XY views hold from 1 to 256 channels of (x, y) co-ordinate pairs that can be displayed in a variety of styles. The following are the XY view specific commands.

MeasureToXY	Create an XY view and associated measurement process
XYAddData	Add data points to a channel of an XY view
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
XYInChan	Count XY points inside polygon defined by a 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
XYOffset	Set individual channel offsets in axis units
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

Vertical cursors The following commands control the vertical cursors.

Cursor	Set or get the position of a cursor
CursorActive	Set or get the active cursor mode
CursorActiveGet	Read back 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
CursorNew	Add a new cursor (at a given position)
CursorOpen	Open cursor dialogs and test if open
CursorRenumber	Renumber the cursors in ascending position order
CursorSearch	Trigger active cursor searches from the script
CursorSet	Set the number (and position) of vertical cursors
CursorValid	Test if an active cursor search succeeded
CursorVisible	Get or set cursor visibility

Text files Spike2 can create, read and write text files. 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 variety of formats, including text
Print	Write formatted output to a file or log window
Read	Extract data from a text file
ReadSetup	Set separators and delimiters for Read and ReadStr
TabSettings	Get or set the tab and indent settings for a text view

Horizontal cursors

Horizontal cursors are controlled from a script with the following commands.

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 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

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
EditFind	Find text
EditPaste	Paste the clipboard into the current text field
EditReplace	Find and replace text
EditSelectAll	Select the entire text or cursor window contents
MoveBy	Move relative to current position
MoveTo	Move to a particular place
Selection$	This function returns the text that is currently selected

Binary files

Spike2 can read and write binary files. Binary files provide links to other programs and are generally more efficient than text for transferring large quantities of data.

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 byte order for read/write of numbers in binary files
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

File system

Spike2 can read file information, change the current directory, delete and copy files and convert foreign files into Spike2 format. You can also execute external files.

FileConvert$	Convert a foreign file to a Spike2 file and open it
FileCopy	Copy one file to a new location
FileDate$	Retrieve date of creation of Spike2 data file
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
FileTime$	Retrieve time of creation of Spike2 data file
FileTimeDate	Retrieve time and date of creation of Spike2 data file as numbers
FileTimeDateSet	Set the time and date of creation/sampling of a data file

User interaction commands

These commands exchange information with the user or let the user interact with data.

Inkey	Return key code of last key user pressed
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
Keypress	Detect if Inkey function would read a character
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
Sound	Play a tone or a .wav file
Speak	Convert text to speech on systems that support this
Yield	Give idle time to the system; delay for a time
YieldSystem	Surrender current time slice and sleep Spike2

You can build dialogs, with a set of user-defined fields:

DlgAllow	Set allowed actions and change and idle functions
DlgButton	Add buttons to the dialog and modify existing buttons
DlgChan	Define a dialog entry as prompt and channel selection
DlgCheck	Define a dialog item as a check box
DlgCreate	Start a dialog definition
DlgEnable	Enable and disable dialog items in change or idle functions
DlgGetPos	Get the dialog position in change and idle functions
DlgGroup	Position a group box in the dialog
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
DlgMouse	Set the mouse position, link user functions to mouse position
DlgReal	Define a dialog entry as prompt and real number input
DlgShow	Display the dialog, get values of fields
DlgSlider	Define a horizontal or vertical slider control
DlgString	Define a dialog entry as prompt and string input
DlgText	Define a fixed text string for the dialog
DlgValue	Gives access to dialog fields in change and idle functions
DlgVisible	Show and hide dialog items in change or idle functions
DlgXValue	Define a dialog entry to collect an x axis value

These commands control the various toolbars and link script functions to the toolbar.

App	Get the view handle of the toolbars
Toolbar	Let the user interact with the toolbar
ToolbarClear	Remove all defined buttons from the toolbar
ToolbarEnable	Get or set the enables state of toolbar buttons
ToolbarMouse	Link script functions to mouse actions
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 toolbar buttons
ScriptBar	Controls the script toolbar buttons

Multiple monitor support

These functions position the Spike2 windows in a multiple monitor environment.

DlgCreate	Position a script-controllable dialog relative to monitors.
System	Get monitor count, positions and identify the primary monitor.
Window	Position a script-controllable window relative to monitors.
WindowVisible	Maximise the application over the entire desktop.

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 an FIR filter in the filter bank
FiltCalc	Force coefficient calculation of an FIR filter in the filter bank
FiltComment$	Get or set the comment for an FIR filter in the filter bank
FiltCreate	Create a new FIR filter definition in the filter bank
FiltInfo	Retrieve information about an FIR filter in the filter bank
FiltName$	Get or set the name of an FIR filter in the filter bank
FiltRange	Get the useful sampling rate range for an FIR 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 FIR coefficients
IIRApply	Apply an IIR filter bank filter to a waveform channel
IIRBp	Create and/or apply an IIR bandpass filter
IIRBs	Create and/or apply an IIR bandstop filter
IIRComment$	Get or set the comment for an IIR filter in the filter bank
IIRCreate	Create a new IIR filter definition in the filter bank
IIRHp	Create and/or apply an IIR highpass filter
IIRLp	Create and/or apply an IIR lowpass filter
IIRName$	Get or set the name of an IIR filter in the filter bank
IIRNotch	Create and/or apply an IIR notch filter
IIRReson	Create and/or apply an IIR resonator filter

Mathematical functions The following mathematical functions are built into Spike2. You can apply many of these functions to real arrays by passing an array to the function.

Abs	Absolute value of a number or array
ATan	Arc tangent of number or array
BetaI	Beta function and incomplete beta function
BinomialC	The binomial coefficient $n!/(k!(n-k!))$
Ceil	Ceiling of a number or array (next highest integral value)
Cos	Cosine of a number or array
Cosh	Hyperbolic cosine or a number or array
Exp	Exponential function of a number or array
Floor	Floor of a number or array (next lowest integral value)
Frac	Remove integral part of a number or array
GammaP	Incomplete Gamma function, used to calculate probabilities
Ln	Natural logarithm of a number or array
LnGamma	Natural logarithm of the Gamma function (use for big factorials)
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	Generate pseudo-random numbers with uniform density
RandExp	Generate random numbers with exponential density
RandNorm	Generate random numbers with normal density
Round	Round a real number to the nearest integral value
Sin	Sine of a number or array
Sinh	Hyperbolic sine of a number or array
Sqrt	Square root of a number or an array
Tan	Tangent of a number or array
Tanh	Hyperbolic tangent of a number or array
Trunc	Remove fractional part of number or array
ZeroFind	Find zero crossings (roots) of a user-defined function

Array and matrix arithmetic

These functions are used with arrays and result views to speed up data manipulation. You can also use the built-in mathematical functions directly on an array.

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 transforms 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
ArrSort	Sort an array and optionally order others in the same way
ArrSpline	Interpolate one array to another using cubic splines
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
LinPred	Linear prediction and maximum emtropy power spectrum
MATDet	Calculate the determinant of a matrix
MATInv	Invert a matrix
MATMul	Multiply matrices and vectors.
MATSolve	Solve a set of linear equations
MATTrans	Transpose a matrix (also see the `trans()` operator)
PCA	Principal component analysis (singular value decomposition)

String functions

These functions manipulate strings and 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 sub-string
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 sub-string of a string
Print$	Produce formatted string from variables
ReadStr	Extract variables from a string
ReadSetup	Set separators and delimiters for `ReadStr` and `Read`
Right$	Returns the rightmost characters of a string
Str$	Converts a number to a string
Trim (Left/Right)	Remove leading/trailing white space or user-defined characters
UCase$	Returns upper case version of a string
Val	Converts a string to number

Curve fitting

These functions do least squares fits of data to functions.

ChanFit	Fit function to time, result or XY channels and display fitted line
FitExp	Fits multiple exponentials
FitGauss	Fit multiple Gaussians (normal distributions)
FitLine	Fit straight line to time or result view data
FitLinear	Fit to linear combination of user-defined functions
FitNLUser	Fit to non-linear user function
FitPoly	Fit polynomial to the data
FitSin	Fit multiple sinusoids

Analysis These functions create new result views and analyse data into the result views.

BinError	Get or set error bar information
MeasureToXY	Create an XY view and associated measurement process
MeasureX	Set the x part of a measurement
MeasureY	Set the y part of a measurement
RasterAux	Set additional data for sorted rasters (deprecated)
RasterGet	Read back raster information
RasterSet	Set raster information
RasterSort	Get and set the sorting values
RasterSymbol	Get and set the time values for raster symbols
SetEvtCrl	Create result view for event correlation
SetEvtCrlShift	Set shift for shuffled event correlation
SetINTH	Create result view for interval histogram
SetPhase	Create result view for phase histogram
SetPower	Create result view for power spectrum
SetPSTH	Create result view for post/peri-stimulus histogram
SetWaveCrl	Create result view for waveform correlation
SetWaveCrlDC	Set DC use/ignore for SetWaveCrl()
SetAverage	Create result view for waveform copy/average/sum
SetResult	Create result view for user-defined data
Sweeps	Number of items added to result view
Process	Process data into result view
ProcessAll	Process all result views attached to a time view
ProcessAuto	Process data automatically during sampling
ProcessTriggered	Process triggered by data during sampling

Arbitrary waveform output These commands control the output of waveforms during data sampling and offline. Waveforms are identified by a key code and up to 10 can be defined.

PlayOffline	Play offline via 1401 or sound card
PlayWaveAdd	Add waveform to the on-line play list
PlayWaveChans	Get or change the DAC channels of an area before sampling
PlayWaveCopy	Update output waveforms in 1401 after sampling starts
PlayWaveCycles	Get or change the repeats of a waveform
PlayWaveDelete	Delete one or more waveforms from the play wave list
PlayWaveEnable	Controls which waveforms in the list are available for playing
PlayWaveInfo$	Get waveform information including list of all waveform keys
PlayWaveKey2$	Associate an additional key with an area after sampling starts
PlayWaveLabel$	Get or set the label associated with each waveform
PlayWaveLink$	Get or set the links between waveforms
PlayWavePoints	Change the size of an area after sampling starts
PlayWaveRate	Get or set the desired base wave replay rate
PlayWaveSpeed	Scale factor for the wave replay rate - can be used on-line
PlayWaveStatus$	Get information about waveform output during sampling
PlayWaveStop	Stop the current output immediately or set one more cycle
PlayWaveTrigger	Get or set the trigger state for a waveform

Debugging operations These functions can be used when debugging a script.

Debug	Set a permanent break point and disable/enable debugging
DebugList	List internal Spike2 script objects
DebugOpts	Gets and optionally sets system level debugging options
Eval	Convert the argument to text and display

Sampling configuration

These commands set the sampling configuration to use when you create a new data file.

SampleAutoComment	Control automatic prompt for file comment after sampling
SampleAutoCommit	Controls how frequently a data file is flushed to disk
SampleAutoFile	Set if file automatically written to disk at end of sampling
SampleAutoName$	Set file name template for automatic file saving
SampleBigFile	Set or get the big file setting in the sampling configuration
SampleCalibrate	Set calibration of waveform or WaveMark channel
SampleChanInfo	Get information about channels in the sampling configuration
SampleChannels	Set or get the maximum number of channels to sample
SampleComment$	Set or get channel comments or Script Bar comment and label
SampleConfig$	Get the name of the last loaded configuration file
SampleDebounce	Get or set the event or digital marker debounce period
SampleClear	Sets the Sampling configuration to a known state
SampleEvent	Adds a channel to sample event data
SampleDigMark	Adds a channel to sample digital marker data
SampleKeyMark	Restore keyboard marker channel to sampling configuration
SampleLimitSize	Set or clear the size limit on a file
SampleLimitTime	Set or clear the time limit for sampling
SampleMode	Set the sampling mode (Continuous, Timed, Triggered)
SampleOptimise	Set methods for optimising waveform and WaveMark rates
SampleRepeats	Set the number of files to sample in sequence
SampleSeqClock	Get and optionally change the sequencer clock rate
SampleSeqCtrl	Set or get the allowed sequencer jump methods
SampleSequencer	Set the name of the sequencer file
SampleSequencer$	Get the name of the sequencer file
SampleStartTrigger	Set triggered or non-triggered sample start
SampleTalker	Add a talker channel to the sampling configuration
SampleTextMark	Adds a channel for text notes
SampleTimePerAdc	Set the base ADC conversion interval
SampleTitle$	Set and get the title of a channel
SampleTrigger	Enable peri-trigger sampling of groups of channels
SampleUsPerTime	Set and get the basic timing interval
SampleWaveform	Adds a channel to sample waveform data
SampleWaveMark	Adds a channel to sample WaveMark data

Runtime sampling

These commands control data sampling.

SampleAbort	Stop sampling and throw data away
SampleHandle	Gets the view handle of sampling windows and controls
SampleKey	Adds to the keyboard marker channel, controls output sequencer
SampleReset	Clear the current data file and restart sampling
SampleSeqClock	Get and optionally change the sequencer clock rate
SampleSeqStep	Get the current sequencer step
SampleSeqTable	Get and set output sequencer table data
SampleSeqVar	Set an output sequencer variable
SampleStart	Start sampling after creating a new time view
SampleStatus	Get the sampling state
SampleStop	Stop sampling and keep the data
SampleText	Adds a string to the text marker channel
SampleWrite	Control writing data to sampling file

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

Environment

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

App	Get the program serial number
Date$	Get system date in a string in a variety of formats
Error$	Convert a runtime error code to a message string
Profile	Access to the system registry and to Spike2 Preferences
ProgKill	Terminate a process started by ProgRun
ProgRun	Run a program, optionally position the window
ProgStatus	Test if a program started by ProgRun is still active
ScriptRun	Set a script to run when the current script ends
Seconds	Get or set current relative time in seconds
Sound	Play a tone or a .wav file
System	Get system revision as number
System$	Get system name as a string, access environment variables
Time$	Get system time in a string in a variety of formats
TimeDate	Get system date and time as numbers

Spike shape window

These functions can be used with a spike shape window.

Cursor	Moving cursor 0 sets the time to read spikes from
FileClose	Close the spike shape window
HCursor	Set the horizontal cursor positions
Optimise	Optimise the y range and horizontal cursors
SSButton	Set and read button states in the window
SSChan	Set and get channel information, save channel configuration
SSClassify	Classify and create WaveMark data
SSColApply	Apply collision analysis result
SSColArea	Get and set the Inportant area for collision analysis
SSColInfo	Get collision analysis information
SSOpen	Open spike shape windows and get handle of open window
SSParam	Equivalent to the spike shape parameters dialog
SSRun	Get and set the run state and speed
SSTempDelete	Delete templates based on index and code
SSTempGet	Read back template shape
SSTempInfo	Get and set template information (width, locking, counts)
SSTempSet	Create and modify templates
SSTempSizeGet	Get template size and offset in display and displayed points
SSTempSizeSet	Set template size and position and display size in points
ViewLink	Get the associated time view handle
Window	Set the position of the spike shape window

WindowGetPos	Get the window position
WindowVisible	Show and hide the spike shape window
XHigh	Get the start of the time view range to process
XLow	Get the end of the time view range to process
XRange	Set time range in associated time view to process
YHigh	Read back the y axis range
YLow	Read back the y axis range
YRange	Set displayed y range

Spike Monitor These function control the Spike monitor window.

SMControl	Get and set the states of the control bar items
SMOpen	Open and or get the handle of a spike monitor window
ViewLink	Get the associated time view handle
Window	Set the position of the spike shape window
WindowGetPos	Get the window position
WindowVisible	Show and hide the spike shape window

Multimedia files These function control avi files associated with data fles.

MMAudio	Get information on the audio content
MMFrame	Get frame positions
MMImage	Get a video image in a variety of formats
MMOffset	Get and set the time offset of the multimedia file
MMOpen	Open and or get the handle of a multimedia file
MMPosition	Set the play position and play state of a multimedia file
MMRate	Set the desired frames per second during recording
MMVideo	Get information about the video format

Serial line control These functions read to and write from serial line ports.

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

1401 control These functions communicate with a 1401 when Spike2 is not sampling.

U1401Close	Release the 1401 for use by Spike2 sampling.
U1401Ld	Load 1401 commands into the 1401 from disk files.
U1401Open	Open a 1401 for use by the other U1401xxx commands.
U1401Read	Read 1401 response, optionally convert to numbers.
U1401To1401	Transfer an integer array to the 1401 from the host.
U1401ToHost	Transfer an integer array to the host from the 1401
U1401Write	Write a text string to the 1401

Discriminator control The Discrim… commands support the obsolete 1401-18 event discriminator card for the 1401*plus*. See the online help and TechNote.pdf in the Spike2 extra Documentation.

-5   Alphabetical command reference

Alphabetical command reference

This section of the manual lists commands alphabetically. If you are not sure which command you require, look in the previous chapter, *Commands by function*. 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 form the absolute value of a real or integer array.

```
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()

This returns view handles for system specific windows and program information.

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

type%   Negative values return system information or 0 if unimplemented:

-1 Spike2 version times 100	-4 Used system handles
-2 The highest type% (10)	-5 Used system graphical objects
-3 The program serial number	-6 1 if 64-bit Spike2, 0 for 32-bit

The remaining values return view handles. If type% is omitted, 0 is used.

0 Application	4 Edit toolbar	8 Sample control panel
1 Main toolbar	5 Play wave bar	9 Sequence control panel
2 Status bar	6 Script bar	10 Sample Status bar
3 Running script	7 Sample bar	

Returns   The requested information or a handle for the selected window. If the requested window does not exist, the return value is 0.

See also: View(), Dup(), LogHandle(), System(), ViewList(), Window()

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 result 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([],fred[10]);         'Add fred[10] to result channel 1
ArrAdd(two[][], add[][]);    'add corresponding elements
```

See also: ArrConst(), ArrDiff(), ArrDiv(), ArrDivR(), ArrDot(), ArrFFT(), ArrFilt(), ArrIntgl(), ArrMul(), ArrSub(), ArrSum(), Len()

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*] := b2d[*j*][*i*]

See also:ArrAdd(),ArrDiff(),ArrDiv(),ArrDivR(),ArrDot(),ArrFilt(),
        ArrIntgl(),ArrMul(),ArrSub(),ArrSubR(),ArrSum(),Len()

ArrDiff()

This function replaces an array or result view with an array of differences. You can use this as a crude form of differentiation, however ArrFilt() provides a better method.

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

dest[] A real or integer vector that is replaced its differences. The first element of dest is not changed.

The effect of the ArrDiff() function can be undone by ArrIntgl(). The following block of code performs the same function as ArrDiff(work[]):

```
var work[100],i%;
for i%:=99 to 1 step -1 do work[i%] -= work[i%-1]; next;
```

See also:ArrAdd(),ArrConst(),ArrDiv(),ArrDivR(),ArrDot(),ArrFFT(),
        ArrFilt(),ArrIntgl(),ArrMul(),ArrSub(),ArrSubR(),ArrSum()

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*] / b2d[*j*][*i*]

See also:`ArrAdd()`,`ArrConst()`,`ArrDiff()`,`ArrDivR()`,`ArrDot()`,`ArrFFT()`,
`ArrFilt()`,`ArrIntgl()`,`ArrMul()`,`ArrSub()`,`ArrSubR()`,`ArrSum()`

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*] / b2d[*j*][*i*]

See also:`ArrAdd()`,`ArrConst()`,`ArrDiff()`,`ArrDiv()`,`ArrDot()`,`ArrFilt()`,
`ArrIntgl()`,`ArrMul()`,`ArrSub()`,`ArrSubR()`,`ArrSum()`,`Len()`

ArrDot()

This function multiplies a vector by another and returns the sum of the products (sometimes called the dot product). The vectors are not changed.

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

arr1   A real or integer vector.

arr2   A real or integer vector.

Returns The function returns the sum of the products of the corresponding elements of the two vectors.

See also:`ArrAdd()`,`ArrConst()`,`ArrDiff()`,`ArrDiv()`,`ArrDivR()`,`ArrFFT()`,
`ArrIntgl()`,`ArrMul()`,`ArrSub()`,`ArrSubR()`,`ArrSum()`,`Len()`

<table>
<tr><td>

ArrFFT()

</td><td>

This command performs spectral analysis on a result view, or 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. The command has the syntax:

</td></tr>
</table>

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

dest   A real vector to process. It should be a power of two points long, from 8 points upwards; the upper size is limited by available memory. 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.

This command uses real arithmetic to calculate the fast Fourier transform. This is more precise, but somewhat slower than the integer transform used by `SetPower()`.

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 result views The phase information sits rather uncomfortably in a result 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([1:],[n/2+1:]); 'Copy phase to the start of the view
[0]:=0; [n/2]:=0;        'Set phase of the DC and Nyquist points
Draw(0,n/2+1);           'Display the phase
```

Mode 0: Window 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 would most likely use this to allow you 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)	DC amplitude	4	4/(n*t)	Nyquist 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), the Nyquist frequency, as this sine wave would have 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). This is because the phase for both of these points is zero. If you want the phase in degrees, multiply by 57.3968 (180º/π). The log amplitude is calculated by taking the result of a forward FFT (same as mode 1 above) and forming:

$dB = 10.0 \, \text{Log}(power)$ The *power* is calculated as 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:

$amplitude = (\cos^2 + \sin^2)^{0.5}$ There is no sin component for the DC and Nyquist

Mode 5: Power and phase This mode produces the same output format as modes 2 and 3, but with the result as power. The sum of the power components is equal to the sum of the squares of the original data divided by the number of data points. 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
$power = DC^2$ or $Nyquist^2$ 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 1 in view 1, and do the following:

```
var spView%, afView%;            'assume in a time view
spView% := SetPower(1,1024);     'select power spectrum
Process(0,1024*View(-1).Binsize(1)); 'process first 1024 points
WindowVisible(1);                'make window visible
afView% := SetAverage(1,1024,0);'To copy 1024 points
Process(0,0);WindowVisible(1);   'show the data
View(afView%);                   'Move to the view holding data
ArrFFT([], 0);ArrFFT([], 5);     'Apply window, take power spectrum
Draw(0,500);Optimise(0);         'Show 500 bins of Power
View(spView%);                   'Look at SetPower() result
Draw(0,500);Optimise(0);         'Show same bins of power spectrum
```

The results are identical except that the ArrFFT() view is 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.

See also:ArrAdd(), ArrConst(), ArrDiff(), ArrDiv(), ArrDivR(), ArrDot(),
        ArrFFT(), ArrFilt(), ArrIntgl(), ArrMul(), ArrSub(), ArrSubR(),
        ArrSum(), Len(), SetPower()

ArrFilt()

This function applies a FIR (Finite Impulse Response) filter to a real array. You can use `FiltCalc()` and `FIRMake()` to generate filter coefficients for a wide range of filters.

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

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

`coef[]` A real vector of filter coefficients. This 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.
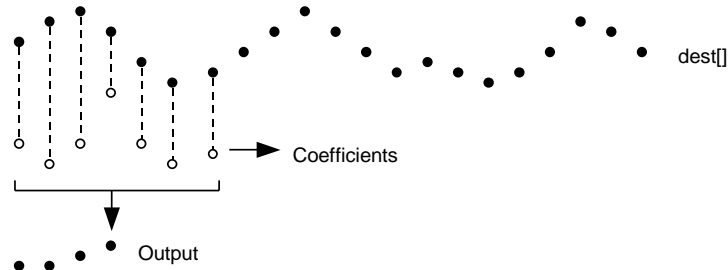
This diagram shows how the FIR filter works. Open circles represent filter coefficients, solid circles are the input and output waveforms. Each output point is generated by multiplying the waveform by the coefficients and summing the result. The coefficients are then moved one step to the right and the 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 in the output. If the filter has an even number of coefficients, there is an output time shift of half a sample.

The filter operation is applied to every vector element. There is a problem at the start and end of the vector where some coefficients have no corresponding data element.
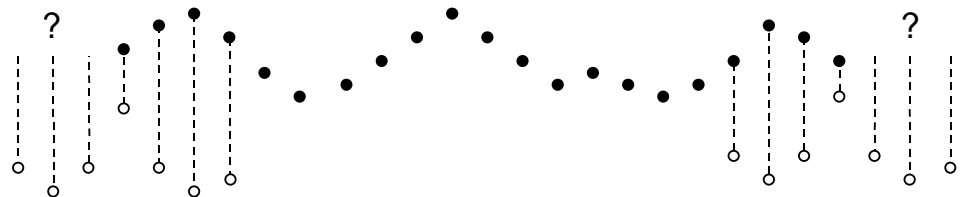
The simple solution is to take these missing points as copies of the first and last points. This is usually better that taking these points as 0. You should remember that the first and last (*nc*+1)/2 points are unreliable, where *nc* is the number of coefficients.

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
coef[0]:=0.33333; coef[1]:=0.33333; coef[2]:=0.33333;
ArrFilt(data[],coef[]);        'smooth the data.
```

If you use this command to apply a causal filter, that is, one with all coefficients that use data points ahead of the current point set to zero, you must still provide these coefficients. If you omit the trailing zero coefficients, the output will be time shifted backwards by half the number of coefficients you do supply.

See also:`ArrDiff(),ArrDiv(),ArrDot(),ArrFFT(),ArrIntgl(),ArrMul(),`
`ArrSub(),ArrSubR(),ArrSum(),FiltCalc(),FIRMake(),Len()`

ArrIntgl()

This function is the inverse of `ArrDiff()`, replacing each point by the sum of the points from the start of the array up to the element. The first element is unchanged.

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

dest   A vector of real or integer data.

The function is equivalent to the following:

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

See also:`ArrAdd()`,`ArrConst()`,`ArrDiff()`,`ArrDiv()`,`ArrDivR()`,`ArrDot()`,
        `ArrMul()`,`ArrSub()`,`ArrSubR()`,`ArrSum()`,`Len()`

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`] * b2d[`j`][`i`]`

See also:`ArrAdd()`,`ArrConst()`,`ArrDiff()`,`ArrDiv()`,`ArrDivR()`,`ArrDot()`,
        `ArrIntgl()`,`ArrSub()`,`ArrSubR()`,`ArrSum()`,`Len()`,`MATMul()`

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.

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. It can also interpolate a matrix of (x,y) value pairs into waveform sampled at a constant interval. Cubic splines assume that the source data has continuous first and second derivatives and that the second derivatives vary linearly. 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.

```
Func ArrSpline(dest[], srcY[]{, xInc{, xStart}});
Func ArrSpline(dest[], srcXY[][]{, xInc{, xStart}});
```

dest   A real or integer vector to hold the interpolated result.

srcY   A vector of y values that is integer if `dest` is integer or real if `dest` is real. The x values are assumed to be the same as the array indices (0, 1, 2...).

srcXY   A matrix of (x,y) values; `srcXY[][0]` is the x, `srcXY[][1]` the y values. The x values must increase with the index or the function returns -1.

xInc   If present, this is the x sampling interval of `dest`. If omitted, we assume that the first and last points of the two arrays are aligned. There are no restrictions on `xInc`; if negative, the output is backwards relative to the input.

xStart If present, this is the x position of the first point of `dest`. If omitted, 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 (for equally spaced source data). There is rarely the need to supply more than 15 data points before and after the interpolation range.

See also: `ChanProcessAdd()`, `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[][]);`	$a2d[j][i] := a2d[j][i] - b2d[j][i]$

See also: `ArrAdd()`, `ArrConst()`, `ArrDiff()`, `ArrDiv()`, `ArrDivR()`, `ArrDot()`, `ArrIntgl()`, `ArrMul()`, `ArrSubR()`, `ArrSum()`, `Len()`

| **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[][]);`	$a2d[j][i] := b2d[j][i] - a2d[j][i]$

See also:ArrAdd(), ArrConst(), ArrDiff(), ArrDiv(), ArrDivR(), ArrDot(), ArrIntgl(), ArrMul(), ArrSub(), ArrSum(), Len()

| **ArrSum()** |

This function forms the sum of the values in a vector or matrix, and optionally forms the mean and standard deviation of the vector or of each matrix column. The standard deviation of m data points with a sum of squared difference from the mean of errSq is sqrt(errSq/(m-1)). There are two command variants:

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

arr   A real or integer vector or matrix 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. If arr is an m by n matrix, mean must be a vector of at least n elements and is returned holding the mean of each column of arr.

stDev If present, this returns the standard deviation of the array elements around the mean. If the array has only one element, the result is 0. If arr is a m by n matrix, stDev must be a vector with at least n elements and is returned holding the standard deviation of each column of arr.

Returns The function returns the sum of all the array elements as a real number.

See also:ArrAdd(), ArrConst(), ArrDiff(), ArrDiv(), ArrDivR(), ArrDot(), ArrIntgl(), ArrMul(), ArrSub(), ArrSubR(), Len()

| **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 calculates the arc tangent of an expression, or the arc tangent of an array:

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

s   A value or an array of real values with any number of dimensions.

c   Optional. If present, both s and c must be values, or both must be real arrays with the same number of dimensions. s/c is a tangent and the result is in the range -π to π using the signs of s and c to determine the quadrant. If c is omitted, the result is in the range -π/2 to π/2, equivalent to c being unity.

Returns If s is an array, each element of s (that is matched by an element of c if c is present) is replaced by the arc tangent. It returns 0 or a negative error code. If s is a variable, the return value is the arc tangent of s or of s/c if c is present.

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

BetaI()

This function calculates both the beta function and the incomplete beta function. The beta function and incomplete beta functions are defined as:

$$Beta(a,b) = \int_0^1 t^{a-1}(1-t)^{b-1}dt \qquad BetaI_x(a,b) = \frac{\int_0^x t^{a-1}(1-t)^{b-1}dt}{Beta(a,b)}$$

These functions are not usually directly of interest, but they are used in generating distribution functions, for example Student's distribution, the F-Distribution and the Binomial distribution. The on-line help has script examples to implement these.

```
Func BetaI(a, b{, x});
```

a, b   These values must be greater than zero.

x   Optional. If present, the result is the incomplete beta function. If omitted, the result is the beta function. x must be in the range 0 to 1.

Returns If x is present, the result is the incomplete beta function in the range 0 to 1. Otherwise, the result is the beta function.

See also:GammaP(), GammaQ()

BinError()

This function is used in a result view to access the optional error (standard deviation about the mean) and bin counts (number of items added to each bin) arrays. Errors are enabled for a result view created with SetAverage() or SetResult() with 4 added to flags%; enable bin counts by adding 32 to flags%. To set a non-zero error the sweep count must be greater than 1. Use Sweeps() to set the count. The first command variant transfers data for a single bin, the others transfer an array of bins:

```
Func BinError(chan%, bin%{, newSD{, newCt%}});
Func BinError(chan%, bin%, sd[]{, set%{, Ct%[]});
Func BinError(chan%, bin%, 0, set%, Ct%[]});
```

chan%   The channel number in the result view.

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

newSD   If present, this sets the standard deviation for bin bin%.

newCt% If present, this sets the item count (if item counts are enabled) for bin bin%.

sd[]   An array for standard deviations or 0 for no transfer. Transfers start at bin bin% in the result view. If the array is too long, extra bins are ignored. If there is no error information, setting has no effect and reading fills sd[] with 0.

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

Ct%[]   If present, used to transfer item count information. If there is no item count, setting has no effect and reading fills the array with the current sweep count.

Returns The first command variant returns the standard deviation at the time of the call. The other variants returns the number of bins copied. If there are 0 or 1 sweeps of data, the standard deviation set will be zero.

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%] next; 'form sum
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 due to calculating the mean from the data. If the result view has item counts per bin enabled (add 32 to flags% in SetAverage() or SetResult()) then replace Sweeps() by the item count for the bin.

See also: BinSize(), BinToX(), SetAverage(), SetResult(), Sweeps()

BinomialC()

This function calculates the binomial coefficient $_nC_k$, which is the number of ways to choose a set of k items from a set of n. The number can be very large as it is given by:

$$\frac{n!}{k!(n-k)!}$$ which can be tricky to compute efficiently, even for a moderate n.

```
func BinomialC(n%, k%);
```

n%   The number of items to choose from. n% must be greater than 0.

k%   The number of items to choose, in the range 0 to n%.

Returns The result, which can be floating point infinity once n% exceeds 1029.

See also: BetaI(), LnGamma()

Binsize()

In a result view, this returns the x axis increment per bin. In a time view, the value returned depends on the channel type.

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

chan%   In a time view this is the channel from which to return information. If you omit the argument, the function returns the file time resolution in seconds. In a result view, chan% is ignored, and should be omitted.

Returns In a time view, the sampling interval between points is returned for Waveform, WaveMark and RealWave channels or a negative number if the channel does not exist. Otherwise, the underlying time resolution of the file in seconds is returned.

See also: BinToX(), XToBin()

BinToX()

This converts bin numbers to x axis units in the current result view. If the current view is a time view, it converts the underlying Spike2 time units into time in seconds.

```
Func BinToX(bin);
```

bin   A bin number in the result view. You can give a non-integer bin number without error. If you give a bin number outside the result view, the bin number is limited to the range of the result view before it is converted to an x axis value.

The x axis range of a result view is `BinToX(0)` to `BinToX(MaxTime())`. Do not confuse this range with `XLow()` to `XHigh()`, which is the visible range of the x axis in the current view.

In a time view, this is in the underlying time units. If the value is beyond the x axis range, it is limited to the x axis range. The value need not be integral, but you should note that all data items in the time view have time stamps that correspond with integral values of `bin`. The returned value is in seconds.

Returns It returns the equivalent x axis position.

See also:`BinSize()`,`MaxTime()`,`XHigh()`,`XLow()`,`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   Arguments may be of any type. Spike2 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 (zero bytes 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   The 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 file opened by `FileOpen()` with a `type%` code of 9. 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()`

BurstMake()

This function extracts burst start and end times from an event channel and writes them to a memory channel. Three parameters control the formation of bursts: the interval between the first two events in a burst, the interval between the last event in a burst and any following event, and the minimum number of events in a burst.
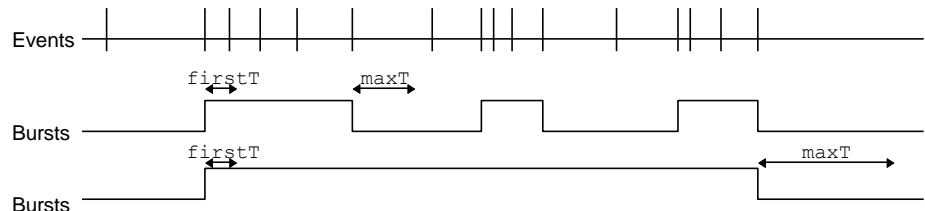
The diagram illustrates the method of forming bursts. In the first case, with a short maximum interval between events, the algorithm finds three bursts. In the second case with a longer period, the algorithm detects one burst. The command for this function is:

```
Func BurstMake(mChan%,eChan%,sTime,eTime,maxT{,firstT{,minE%}});
```

mChan% The channel number of an event or marker channel created by `MemChan()` for the output. If this is a marker channel, the start of each burst is given a first marker code of 00 and the end has a first marker code of 01.

eChan% The channel number of an event channel to search for bursts.

sTime   The start of the time range to search for bursts.

eTime   The end of the time range to search for bursts.

maxT   The maximum time between two events (after the first pair) for the events to lie in the same burst.

firstT   The maximum time between the first two events in a burst. If omitted, maxT sets the time interval for the first pair of events.

minE%   The minimum number of events that can make up a burst. The default is 2.

Returns   The function returns the number of bursts found, or a negative error code.

See also: BurstRevise(), BurstStats(), MemChan()

BurstRevise()

This command modifies a list of times, indicating bursts and produces a new list of bursts that have inter-burst intervals and burst durations greater than specified minimum times (See BurstStats() for more information).

```
Func BurstRevise(mChan%, eChan%, sTime, eTime, minI, minD);
```

mChan%   A memory channel created by MemChan() holding event data to which the output of this command is added. This can be the same channel as eChan%, but if it is, the output is generated by deleting unwanted events from the channel.

eChan%   The event or marker channel to read burst times from.

sTime   The start of the time range to revise.

eTime   The end of the time range to revise.

minI   The minimum interval between the end of one burst and the start of the next. Bursts with shorter intervals are amalgamated.

minD   The minimum duration of a burst. Shorter bursts are deleted.

Returns   The number of bursts in the output or a negative error code.

See also: BurstMake(), BurstStats(), MemChan()

BurstStats()

This command returns statistics on bursts (possibly made by BurstMake()). Additional rules can be applied to the bursts before the statistics are calculated to amalgamate bursts that are too close together and to delete bursts that are too short. The statistics are the mean and standard deviation of the duration of the bursts and the intervals between the start of one burst and the start of the next.
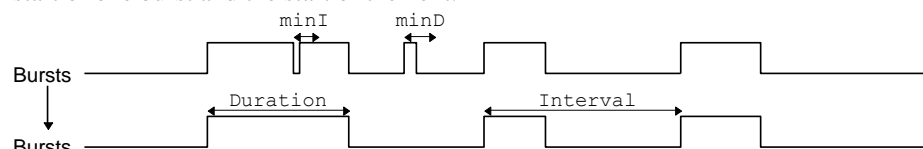

```
Func BurstStats(eChan%, sTime, eTime,
                        &meanI, &sdI, &meanD, &sdD{,minI{,minD}});
```

eChan%   The event channel containing burst data. The first event found in the time range is assumed to be the start of a burst, the second the end, and so on.

sTime   The start of the time range to process.

eTime   The end of the time range to process.

meanI   This is a real variable that is returned holding the mean interval between the starts of bursts (as long as at least 2 bursts were found).

sdI   This is a real variable that is returned holding the standard deviation of the mean interval (as long as at least 3 bursts were found).

meanD   This is a real variable that is returned holding the mean burst duration (as long as at least 1 burst was found).

sdD   This is a real variable that is returned holding the standard deviation of the burst durations (as long as at least 2 bursts were found).

minI   This optional value sets the minimum interval between the end of one burst and the start of the next. It takes the value 0.0 if omitted. Bursts that are closer than this are amalgamated for the purpose of forming statistics.

minD   This optional value (taken as 0.0 if omitted) sets the minimum burst duration. Bursts shorter than this (after amalgamations) are ignored in the statistics.

Returns   It returns the number of bursts used for the statistics, or a negative error code.

This command is used in scripts that optimise the values of `minI` and `minD` for forming bursts. For example, when bursting is known to follow a cyclical pattern, one would like to find values of `minI` and `minD` that minimise the total coefficient of variance:

Total coefficient of variance $= (\mathtt{sdD}/\mathtt{meanD})^2 + (\mathtt{sdI}/\mathtt{meanI})^2$

Once suitable values are found, the `BurstRevise()` command can generate a memory channel holding bursts based on the optimised parameters.

See also: `BurstMake()`, `BurstRevise()`, `MemChan()`

BWrite()

This function writes binary data values and arrays into a file opened by `FileOpen()` with a `type%` code of 9. 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   Arguments may be of any type, including arrays. Spike2 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 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 function writes variables or arrays as binary into a file opened by `FileOpen()` with a `type%` code of 9. It writes 8, 16 and 32-bit integers and 32 and 64-bit reals and strings. It allows you to write formats other than the 32-bit integer and 64-bit real used internally by Spike2 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 of `size%` 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   The 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 0 or a negative error code for an array or the next higher integral value.

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. Memory, duplicate and virtual channels are listed as they appear on screen (not just as numbers). 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,m2,v1a"`.

See also:`ChanList()`

ChanCalibrate()

This function is equivalent to the Analysis menu Calibrate command. It changes the scale and offset of Waveform and WaveMark channels and rewrites RealWave channels so that user-defined data sections have user-defined values. The command is:

```
Func ChanCalibrate(cSpc,mode%,cfg%,t1,t2,units$,v1{,v2{,t3,t4}});
```

cSpc   A channel specifier for the channels to calibrate or -1 for all, -2 for visible and -3 for selected channels. You cannot calibrate processed channels if the process changes the scale or offset or any processed RealWave channel.

mode%   The calibration mode. The items in brackets are the required optional arguments.
   0   Mean levels of two time ranges (`v2`, `t3`, `t4`).
   1   Values at two time points (`v2`).
   2   Set offset from mean of time range.
   3   Set scale from mean of time range.
   4   Square wave, upper and lower level (`v2`).
   5   Square wave, amplitude (Size) only.
   6   Peak to peak amplitude and mean (`v2`).
   7   RMS amplitude about mean (`v2`).
   8   Area under curve, assume zero at end.
   9   Areas under curve, two time ranges (`v2`, `t3`, `t4`).
   10   Slope of best-fit line to the data, the offset not changed.

cfg%   If non-zero, the new calibration is saved in the sampling configuration.

t1,t2   In all modes except mode 1, these are the start and end of the first time range. In mode 1, these times correspond to the two calibration values `v1` and `v2`.

units$ The units to apply to the channel.

v1,v2   The values in user units that correspond to the times or time ranges. v1 is always used; v2 is only used in modes 0, 1, 4, 6, 7 and 9.

t3,t4   A second time range. These values are used in modes 0 and 9.

Returns   The return value is the sum of the following values:
  1 A channel in the list did not exist or was the wrong type.
  2 A channel was processed OK.
  4 Unknown or unimplemented calibration mode.
  8 A time range had the end past the start or two time ranges overlapped or the two times in mode 1 were the same.
  16 The v1 and/or v2 values were too big, too small or too similar.
  32 Not enough data to process at least one channel in the list.
  64 The data is unsuitable. For example, in mode 0 mean levels must differ by at least the standard deviation of the data around the mean.

See also:ChanOffset(), ChanScale(), ChanUnits$()

ChanColour()

Returns and optionally sets a channel colour from the palette in a time or result view to override the application colour set for the drawing mode of the channel.

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

chan%   A channel in the time or result view. The channel must exist.

item%   The colour item to get or 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 nearest palette colour index to the item colour 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(), ViewColour(), XYColour()

ChanColourGet()

Returns a channel item RGB colour in a time, result or XY view. Added at version 7.07.

```
Func ChanColourGet(chan%, item%{, &r, &g, &b});
```

chan%   A channel in the time, result or XY view.

item%   The colour item to get; 0=background, 1=primary, 2=secondary/fill colour.

r g b   If present, get the colour as red, green and blue values in the range 0 to 1.0.

Returns   1 if the channel colour is overridden, 0 if not, negative for no channel.

See also:ColourGet(), ViewColourGet()

ChanColourSet()

Sets a channel item RGB colour in a time, result or XY view. Added at version 7.07.

```
Func ChanColourSet(chan%, item%{, r, g, b});
```

chan%   A channel in the time, result or XY view.

item%   The colour item to set; 0=background, 1=primary, 2=secondary/fill colour.

r g b   If present, set the item colour as a red, green and blue value limited to the range 0 to 1.0. If omitted, the item colour is no longer overridden.

Returns   0 or a negative error code if the channel does not exist.

See also:ColourGet(), ViewColourGet()

ChanComment$()

This returns or sets the comment string for a channel in a time or result view; use in any other view type is an error.. The comment can be up to 71 characters long in a time view. In a result view, comments are truncated to 79 characters if the result view is saved.

```
Func ChanComment$(chan%{, new$});
```

chan%   A channel in the time view. In a result view, if this is a duplicate channel, the comment for the original channel is returned and set.

new$ An optional string with a new comment. If the string is too long, it is truncated.

Returns It returns the comment string for the designated channel. If the channel does not exist, the function does nothing and returns an empty string.

See also:ChanTitle$(), FileComment$()

ChanData()

Fills an array with waveform data from a waveform or RealWave channel or with event times from all other channel types. Use NextTime() and LastTime() to get data attached to WaveMark, RealMark and TextMark channels and marker codes.

```
Func ChanData(chan%, arr[]|arr%[], sTime, eTime {,&fTime});
```

chan%   The data channel in the current time view to read data from.

arr   A real or integer array. Real arrays collect waveform data in user units and event times in seconds. Integer arrays collect waveforms in ADC units and event times in the underlying time units (as returned by Binsize()).

sTime   The first data value returned is at or after this time in seconds.

eTime   The last data point returned is before or at this time in seconds.

fTime   This optional argument is a variable that is set to the time of the first data point.

Returns The number of data values placed in the array. Only contiguous waveform data is returned; gaps terminate a read.

See also:Binsize(), ChanValue(), ChanWriteWave(), NextTime(), ChanScale()

ChanDelete()

This deletes a channel from a time, result or XY view. You can make the user confirm time view channel deletion if the channel is stored in the file. Duplicated channels, and memory buffer channels are not confirmed. In a result view, you may only delete duplicate channels. In an XY view, you cannot delete the last XY channel as XY views must always have at least one channel. Channels are always numbered consecutively in an XY view, so if you delete a channel, the channel numbers of any higher numbered channels will change.

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

cSpc   A channel specifier for the channels to delete or -1 for all, -2 for visible or -3 for selected. In an XY view, only channel numbers greater than 0 are allowed.

query%   If present and non-zero, the user is asked to confirm the deletion if the channel is part of a time view. You cannot delete channels that are being sampled.

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:ChanDuplicate(), MemChan(), XYDelete(), XYSetChan()

ChanDuplicate()

This duplicates a time or result view channel. It can also be used in the Edit WaveMark view to delete all duplicates of the current channel and generate duplicates with suitable marker filter setting for each template. Use `DupChan()` to find duplicates of a channel.

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

chan% For time or result views, this is channel number to duplicate; this channel must exist. This argument must be omitted for Edit WaveMark views.

Returns For time or result views it returns the channel number of the duplicate. For an Edit WaveMark view, it returns the number of duplicates that were created. Any error stops the script.

For time and result views, the new channel is not displayed. Use `ChanShow()` to make it visible. In an Edit WaveMark view, channels are made visible immediately. The following example duplicates a time or result view channel and makes it visible:

```
var ch%;ch% := ChanDuplicate(1);   'create a duplicate
ChanShow(ch%);                      'make visible
```

See also:`ChanShow()`,`ChanDelete()`, `DupChan()`, `MemChan()`

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 time, result 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 time, result or XY view can have one fit associated with it.

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

order% The order of the fit. This is 1 or 2 for an exponential or Gaussian fit, 1 for a Sine or Sigmoid fit and 1 to 5 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%, 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.

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 values 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	-9	User-defined x draw start
-2	Type of existing fit or 0	-10	User-defined x draw end
-3	Order of existing fit	-11	1=chi-square, 0=least-square
-4	Chi or least-squares error	-12	Last fit result code
-5	Fit probability (estimated)	-13	Number of fitted points
-6	X axis value at fit start	-14	Number of fit iterations used
-7	X axis value at fit end	-15	R-square value
-8	Reference x value	-16	Adjusted R-square value

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

See also:ChanFitCoef(), ChanFitShow(), ChanFitValue()

ChanFitCoef()

This command gives you access to the fit coefficients for a channel in the current time, result 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 omitted, the return value is the number of coefficients in the current fit, else it is a coefficient number and 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. The first coefficient is 0.

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

lower   If present, 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, sets the upper limit for coefficient num% for the next fit.

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 time, result 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 (if not added, any fit is hidden).
    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=start, -2=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 time, result 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. Some fitting functions can overflow floating point range if x is outside 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()

ChanHide()

Hide a channel, or a list of channels. Hiding a channel that doesn't exist has no effect.

```
Proc ChanHide(cSpc {,cSpc...});
```

cSpc   A channel specifier or -1 for all, -2 for visible, and -3 for selected channels.

See also:ChanShow(), ChanList()

ChanImage()

You can set a background image behind each channel in a Time, Result or XY view (XY view channels share the same bitmap). Images are read from `bmp`, `png`, `jpg` and `tif` files. See the View menu Channel Image command for details.

```
Func ChanImage(chan%, path$);
Func ChanImage(chan%, mode%, opac{, xl, yl, xh, yh});
Func ChanImage(chan%, get%{, &path$});
```

chan%   A channel number in the current view.

path$ The file name that holds the image. The first command variant sets the name, the third variant returns it. If you set an empty name, this releases any memory used to cache the bitmap within the program. Setting an image does not change the display mode; use the second variant to make sure that the image is visible. Use `"<CB>"` as a name to copy a clipboard image.

mode%   There are three display modes: 0=no display, 1=fill background, 2=fill rectangle. You can also set mode -1, meaning no mode change.

opac   You can control the image opacity in the range 0.0 (transparent) to 1.0 (opaque). You can also set the value -1 for no change.

xl-yh   When mode% is 2, these four arguments set the rectangle, in x and y axis units, that contains the bitmap image.

get%   This is used in the third command variant to read back the current settings. You can set -1 to read the mode, -2 to read the opacity, and -3 to -6 to read the `xl`, `yl`, `xh` and `yh` values.

Returns The first variant returns 1 if a bitmap was read, 0 if it was not (either file not found, or no image was set) or a negative error code. The second variant returns 0 and the third variant returns the requested information.

See also:`ChanColour()`

ChanIndex()

Get or set the index associated with a channel in a time view. RealMark channels use the index to select the data value to display.

```
Func ChanIndex(chan%{, index%});
```

chan%   A channel number. Currently, this is useful only with RealMark channels.

index% If present and positive, this sets the zero-based channel index. If omitted or -1, the command returns the current index. If -2, the command returns the number of index values for the channel.

Returns Either the channel index at the time of the call, or if `index%` is -2, the number of possible index values.

See also:`MarkInfo()`

ChanKind()

This returns the type of a channel in the current time, result or XY view.

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

chan%   The channel number.

Returns A code for the channel type or -3 if this is not a time, XY or result view:

0	None/deleted	3	Event (Event+)	6	WaveMark	9	Real wave
1	Waveform	4	Level	7	RealMark	120	XY channel
2	Event (Event-)	5	Marker	8	TextMark	127	Result channel

The result and XY codes changed at version 3.16. Use `ViewKind()` to detect result and XY views if the script must be compatible with all versions of Spike2.

See also:`ChanList()`,`MemChan()`,`ViewKind()`

ChanList()

This function fills an array with a channel list for a time, result or XY view. The result can be filtered to a subset of the available channels. There are two command variants:

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

list%   An integer array to hold channel numbers. Index 0 holds the number of channels returned; the remaining elements are channel numbers. It is unnecessary to list all the channel numbers for an XY view, since they are numbered contiguously.

types%   This specifies the channels to return. If omitted, all channels are returned. The values are the same as those defined for the `DlgChan()` function. XY views only allow exclude visible (1024) and exclude hidden (2048) channels.

str$ A channel specification such as `"1..10,v1,m1"` or `"v1a"`. 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 were of unlimited length or 0 if the view is not the correct type.

See also: `ChanKind()`, `ChanShow()`, `DlgChan()`

ChanMeasure()

This performs the cursor regions measurements on a channel in a time or result window.

```
Func ChanMeasure(chan%, type%, sPos, ePos{, &data%{, kind%}});
```

chan%   The number of the channel to measure.

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 Area	5 Area (scaled)	9 Minimum	13 Abs max.	17 Mean in X
2 Mean	6 Curve area	10 Peak to Peak	14 Peak	18 SD in X
3 Slope	7 Modulus	11 RMS Amplitude	15 Trough	19 Mean of abs
4 Sum	8 Maximum	12 Standard deviation	16 RMS error	

sPos   The start position for the measurement in x axis units. If `sPos` is greater than or equal to `ePos`, the result is 0. In a result view, if using a cursor measurement (`Cursor(1)`, for example) for `sPos` or `ePos`, use `BinToX(Cursor(1))`.

ePos   The end position in x axis units. In a result view, this is converted to a bin number which must be greater than the bin number obtained from `sPos`.

data%   Optional variable returned as 1 if a valid result was obtained or as 0 if there is no result (equivalent to a blank cell in the Cursor Regions dialog).

kind%   This optional variable forces a WaveMark channel to be treated as a waveform or as events for this measurement. If 0 or omitted, the channel is treated as a waveform if drawn in waveform, WaveMark or Cubic Spline mode, otherwise it is treated as an event channel. 1 forces waveform and 2 forces event.

Returns   The function returns the requested measurement value.

See also: `ArrSum()`, `ChanData()`, `ChanValue()`, `XToBin()`

ChanNew()

This function creates a new channel in the current time view. Unlike a memory channel, the created channel is permanent; any data written to it occupies disk space. You can use this to create channels for use with `ChanWriteWave()` and `ChanSave()`.

```
Func ChanNew(chan%, type%{, size%{, binsz{, pre%{, trace%}}}});
```

chan%   The channel number to use in the range 1 to the number of channels in the data file (usually 32) or 0 for the first unused disk-based channel. You cannot use this routine to overwrite an existing channel; use `ChanDelete()` to remove it first.

type%　The type of channel to create. Codes are:

1 Waveform	4 Level (Event+-)	7 RealMark
2 Event (Event-)	5 Marker	8 TextMark
3 Event (Event+)	6 WaveMark	9 Real wave

size%　Used for TextMark, RealMark and WaveMark channels to set the maximum number of characters, reals or waveform points to attach to each item. You can use this to set the disk buffer size in bytes for other channel types, but we recommend that you use 0 for the default buffer size.

binsz　Used for waveform and WaveMark data to specify the time interval between the waveform points. This is rounded to the nearest multiple of the underlying time resolution. If you set this 0 or negative, the smallest bin size possible is set.

If binsz is not a multiple of the underlying time resolution times *time per ADC* for the file, versions of Spike2 before 4.03 cannot read it.

pre%　This must be present for WaveMark data to set the number of pre-trigger points.

trace%　Optional, default 1, the number of interleaved traces for WaveMark data.

Returns　The channel number if the channel was created, or a negative error code.

The channel has default title, units and comment. Use ChanTitle$(), ChanUnits$(), ChanComment$(), ChanScale() and ChanOffset() to set these. The following code creates a copy of channel wFrom% (a waveform channel) in channel wTo%:

```
func CopyWave%(wFrom%, wTo%) 'Copy waveform to a memory channel
var err%, buffer%[8192], n%, sTime := 0.0;
if ChanKind(wFrom%)<>1 then return -1 endif; 'Not a waveform!
if ChanKind(wTo%)<>0 then ChanDelete(wTo%) endif; 'Check if used
ChanNew(wTo%, 1,0,BinSize(wFrom%));      'Create waveform channel
if err%= wTo% then                       'Created OK?
  ChanScale(wTo%, ChanScale(wFrom%));    'Copy scale...
  ChanOffset(wTo%, ChanOffset(wFrom%));  '...and offset...
  ChanUnits$(wTo%, ChanUnits$(wFrom%));  '...and units
  ChanTitle$(wTo%, "Copy");              'Set our own title
  ChanComment$(wTo%, "Copied from channel "+Str$(wFrom%));
  repeat
    n% := ChanData(wFrom%, buffer%[], sTime, MaxTime(), sTime);
    if n% > 0 then                       'read ok?
      n% := ChanWriteWave(wTo%, buffer%[:n%], sTime)
    endif;
    if n% > 0 then sTime += n% * BinSize(wTo%) endif;
  until n% <= 0;
  ChanShow(wTo%);                         'display new channel
endif;
return err%;   'Returns 0 if created OK
end;
```

See also:ChanData(), ChanDelete(), ChanSave(), ChanWriteWave(), MemChan()

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, 1 shows it. Other values are reserved.

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

See also:YAxis(), YAxisMode()

ChanOffset()

Waveform and WaveMark data is stored as 16-bit integers with a scale factor and offset to convert to user units; this function gets and/or sets the y axis value that corresponds to a 16-bit waveform data value of zero. The y axis user units for a channel are:

```
y axis value = (16-bit value)*scale/6553.6 + offset
```

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

chan%   The channel number.

offset If present, this sets the new channel offset. There are no limits on the value.

Returns The old channel offset if this is a waveform or WaveMark channel, or 0 if not.

See also: ChanCalibrate(), ChanScale(), Optimise()

ChanOrder()

This command changes the channel order in time or result views, groups channels with a y axis to share a common y axis, sets the channel sort order and gets channel lists.

```
Func ChanOrder(dest%, pos%, cSpc);                          Move channels
Func ChanOrder(dest%, opt%);                                Group operations
Func ChanOrder(order%);                                     Sorting operations
Func ChanOrder(list%[]{, sel%});                            Get channel list
```

dest%   The destination channel number or channel number used with opt%. When moving channels, dest% values less than 1 refer to the channel position in the list of all channels including hidden channels but excluding channels in cSpc. 0 is the top channel, -1 is the next, and so on. If there are n channels, values of -n or less are taken to mean the bottom channel.

pos%   The drop position relative to the destination channel: -1=drop above, 0=drop on top, 1=drop below. If you drop between grouped channels, the dropped channels become 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% This form of the command sorts all the channels into numerical order. Set -1 for low, 1 for high numbered channels at the top and 0 to use the default channel ordering set by the Edit menu Preferences.

list%   Returned holding a list of visible channels selected by sel%, ordered from top to bottom. list%[0] holds the size of the list; if n, indices 1-n hold the channels.

sel%   Set 0 or omit for all channels, -1 for the first channel of each group, n (greater than 0) for all the channels in group n.

Returns With cSpc it returns the number of moved channels. With order%, it returns -1 if low and 1 if high numbered channels were previously at the top. With list% it returns the number of channels (even if list% is too small to hold them).

See also: ChanWeight(), ViewStandard(), Yaxis(), YaxisLock()

ChanPenWidth()

This command sets and gets the pen width set for a channel in a Time or result view. The pen width for channels in an XY view is handled by the XYDrawMode() command. This script command is the equivalent of the View menu channel Pen Width dialog.

```
Func ChanPenWidth(cSpc{, new});
```

cSpc   A channel specifier for the channels or -1 for all, -2 for visible and -3 for selected channels.

new   If present, the new width of the pen to use for the specified channels, in points. A point is 1/72 of an inch, which is approximately 1 pixel. If you set a negative width, the channel will use the data pen width set in the Edit menu Preferences. A width of zero will set the thinnest pen (1 pixel) possible.

Returns The command returns the current pen width setting for the first channel that exists in the channel specification, or 0 if no channel exists.

ChanPixel()

This gets the x and y axis units that correspond to one pixel on the screen. Normally the x and y axes are in linear mode, but they can be set to logarithmic mode, in which case the returned values are the log increment that corresponds to one pixel on the screen.

```
Func ChanPixel(chan%, &x, &y);
```

chan% The channel in the current Time, Result or XY view.

x   Set to the x axis units equivalent to moving one pixel to the right.

y   Set to the y axis units equivalent to moving one pixel up or 0 if no y axis.

Returns A set of flags for what is returned: 1 if the x axis is set, 2 if the y axis is set, 4 if the x axis is in log mode, 8 if the y axis is in log mode.

ChanPort()

This returns the physical hardware port that sampled data in a time view channel.

```
Func ChanPort(chan%);
```

chan% The channel number in the current time view.

Returns The physical data port or -1 if the view is not a time view, or if the channel was not sampled. Both event and ADC ports are enumerated from 0.

See also:SampleEvent(), SampleWaveform(), SampleWaveMark()

ChanProcessAdd()

This adds a channel process to a waveform or RealWave channel in a time view, matching the effect of the Add button in the Channel Process dialog.

```
Func ChanProcessAdd(chan%, PType% {, arg1, arg2, ...});
```

chan% A waveform or RealWave channel number in the current time view.

PType% The process type. The following process types are currently defined:

0   Rectify. Positive values are unchanged, negative values are negated. Data from waveform channels with a non-zero channel offset may be limited.

1   Smooth. This has one argument, a time range in seconds. The output at time t is the average of the input data from times t-arg1 to t+arg1.

2   DC Remove. There is one argument, a time range in seconds. The output at time t is the input minus the mean input from t-arg1 to t+arg1.

3   Slope. There is one argument, a time range in seconds. The slope at time t is calculated from the points in the time range t-arg1 to t+arg1. If you apply this process, the channel scale and y axis units change and the channel offset becomes 0. The output is in input units per second.

4   Time shift. There is one argument, a time shift in seconds. A positive value shifts the trace right in the window, a negative value shifts it left.

5   Down sample. There is one argument, the down sample ratio.

6   Interpolate. Argument 1 is the sample interval, 2 is the alignment.

7   Chan match. Argument 1 is the channel to match.

8   RMS amplitude. Argument 1 is the time range.

9   Median filter. Argument 1 is the time range.

10 Fill gaps. Argument 1 is the maximum gap to interpolate across, 2 is the fill level for larger gaps.

11 Skip Nan. Removes non-numbers from RealWave data, leaving gaps.

arg#   Optional process arguments to match the Channel Process dialog arguments for each process type. Each process has default arguments. It is an error to provide too many arguments or for an argument to be incorrect for the process.

Returns The index of the added process in the list of processes for the channel.

See also:`ChanProcessArg()`, `ChanProcessClear()`, `ChanProcessInfo()`

ChanProcessArg()

This gives you access to the parameters of channel processes added by the Channel Process dialog or by the `ChanProcessAdd()` command.

```
Func ChanProcessArg(chan%, id% {,n% {,arg}});
```

chan%   The channel number in the current time view.

id%   The process index. The first one added is number 1, the second is 2, and so on.

n%   An optional argument number. The first argument is 1, the second 2, and so on.

arg   If present, changes process argument n%. Ignored if out of range or illegal.

Returns If n% is omitted the function returns the number of arguments process id% uses. If n% is present, it returns the value of argument n% at the time of the call.

See also:`ChanProcessAdd()`, `ChanProcessClear()`, `ChanProcessInfo()`

ChanProcessClear()

This removes one or all processes from a channel or all processes from all channels.

```
Func ChanProcessClear({chan% {, id%}});
```

chan%   A channel number in the current time view. If you omit this argument, or set it to -1, all processes for all channels are removed.

id%   The index of the process to delete. Set -1 to delete all processes for channel chan%. After deleting, any remaining processes are renumbered.

Returns 0 if at least one process was deleted, -1 if no process was deleted.

See also:`ChanProcessAdd()`, `ChanProcessArg()`, `ChanProcessInfo()`

ChanProcessCopy()

This copies all the processing from a source channel to a list of target channels, or clears all processing from a list of channels.

```
Func ChanProcessCopy({cSpc%{, src%}});
```

cSpc   A channel specifier for the target channels or -1 for all, -2 for visible and -3 for selected channels.

src%   If omitted or 0, all processing for the channels defined by cSpc is cleared. Otherwise, it is the channel to copy the processing from.

Returns 0 if at least one process was deleted, -1 if no process was deleted.

See also:`ChanProcessAdd()`, `ChanProcessArg()`, `ChanProcessClear()`

ChanProcessInfo()

This returns information about processes attached to a channel.

```
Func ChanProcessInfo(chan% {,id% {, arg%}});
```

chan%   The channel number in the current time view.

id%   Omit this to return the count of processes for the channel or set it to the index of the process to return information on. The first added process is number 1.

arg%   Omit this to return the type of process index id%; see `ChanProcessAdd()` for the type codes. If present, the command returns the type of argument arg%: 0=integer, 1=real, 2=string (unused), 4=channel. The first argument is number 1.

Returns The information requested or a negative error code.

See also:`ChanProcessAdd()`, `ChanProcessArg()`, `ChanProcessClear()`

ChanSave()

This function copies data with an optional time shift from source channels in the current time view to disk-based or memory buffer destination channels in any time view. If a destination channel exists, `ChanSave()` adds data to it, otherwise it creates a disk-based channel to match the source channel. You can set the number of disk-based channels in a new data file with the nChans% argument of `FileNew()`.

The source data must be compatible with the destination. All event-based data types are mutually compatible. Destination data that is not present in the source is set to 0. Event times are set as close to the original times as the time bases of the source and destination views allow. If the conversion causes multiple events to fall at the same time, the converter adds 1 tick to separate the times. However, it will not do this for more consecutive events that the ratio of the two time bases.

If the source channel has a marker filter set, only events in the source channel that match the marker filter are copied to the destination channel. If the destination channel has a marker filter, the filter is unchanged for an existing channel and is set to accept all data for a new channel.

Adc and RealWave channels are compatible; conversion between them uses the channel scale and offset values. If the source and destination sample rates do not match, the data is interpolated using cubic splines. This is also done for non-matching WaveMark data.

If a destination disk-based channel already contains data, any added event-based data must be written after all data already contained in the channel. You can over-write existing waveform data with the same restrictions as for `ChanWriteWave()`.

```
Func ChanSave(cSpc, dest%{, dh%{, sTime{, sTo{, dTime}}}});
```

cSpc   A channel specifier for the source channels or -1 for all, -2 for visible and -3 for selected channels. If you set multiple channels, dest% must be set to 0 or -1.

dest%   This sets the destination channel or channels in the view identified by dh%. If cSpc refers to a single channel, this can be a channel number of a disk-based channel or an existing memory channel; otherwise, dest% must be 0 or -1.

        If dest% is 0, the lowest unused disk-based channels are used. If dest% is -1, the same channel number as the source is used. If the destination channel does not exist, it is created using the source channel settings.

dh%   The handle of the destination view. Omit or set 0 to use the current time view.

sTime   The start time to read source data. This is 0.0 if omitted.

sTo   Source data is read up to this time. If omitted, the end of the data is used.

dTime   The destination time. If this is omitted, sTime is used. The source data is time shifted by dTime-sTime before being copied to the destination channels.

Returns With a single source channel, the result is the destination channel number. With multiple source channels, the result is the number of channels copied without

error. Negative results are: -1 = cSpc illegal, -2 = dest% illegal, -3 = dh% is not a time view handle, -4 = destination channel could not be created or was not compatible with the source, -5 = a file system error occurred when copying data.

Append files example

This command has many applications. This example creates a new file and adds the contents of two data files to it. We assume the files hold the same types of channels. In this case, we create an output file with the same time resolution as the first input file.

```
var fh%,dh%;     'file handles of the source and destination files
fh% := FileOpen("file1.smr", 0, 0);                  'open file1
dh% := FileNew(7, 1, View(fh%).Binsize()*1e6, 1, 1);  'make output
View(fh%);ChanSave(-1, -1, dh%);FileClose(0,-1);   'copy and close
fh% := FileOpen("file2.smr", 0, 0);          'open file2 and append
ChanSave(-1, -1, dh%, 0, MaxTime(), View(dh%).MaxTime()+1.0);
FileClose(0,-1);                                     'close file2
View(dh%);ChanShow(-1);Draw(0, MaxTime());    'Display the result
```

See also:ChanNew(), ChanOffset(), ChanScale(), ChanWriteWave(), FileNew()

ChanScale()

This function gets and/or sets the scaling between the 16-bit integer data used to store waveform and WaveMark data and the real units of the channel. The 16-bit waveform data has values between +32767 and -32768. The y axis user units for a channel are:

```
y axis value = (16-bit value)*scale/6553.6 + offset
```

With the standard ±5 Volt 1401 ADC inputs, 6553.6 is equivalent to 1 Volt at the 1401 input. In this case, a scale of 1.0 and an offset of 0.0 gives a y axis calibrated in Volts.

```
Func ChanScale(chan% {,scale});
```

chan%   The channel number.

scale   If present, sets the channel scaling. We suggest you don't set 0.0 as a scale!

Returns   The scale at the time of the call for waveform or WaveMark channels, or 0.

See also:ChanCalibrate(), ChanOffset(), Optimise()

ChanSearch()

This searches a time or result view channel for a user-defined feature. It is the same as an active cursor search, but does not use or move cursors. Searches are done in the current channel drawing mode. WaveMark data drawn as a waveform is searched as a waveform; to search it as events, set an event drawing mode. There is no need for the channel to be visible; you only need to set the display mode before searching.

```
Func ChanSearch(chan%, mode%, sT, eT{, sp1{, sp2{, width
                                   {, flag%{, lv2}}}}});
```

chan%   The number of a channel in the time view to search.

mode%   This sets the search mode, as for the active cursors. Modes in italics (15 and 16) cannot be used: mode 16 is not relevant and mode 15 can be emulated by mode 6, 7 or 8. See the cursor mode dialog documentation for details of each mode.

1	Maximum value	8	Falling threshold	*15*	*Repolarisation %*
2	Minimum value	9	Steepest rising	*16*	*Expession*
3	Max excursion	10	Steepest falling	17	Turning point
4	Peak find	11	Steepest slope (+/-)	18	Slope%
5	Trough find	12	Slope peak	19	Outside thresholds
6	Threshold	13	Slope trough	20	Inside thresholds
7	Rising threshold	14	Data points		

sT,eT   The search start and end positions as seconds in a time view or as bins in a result view). If eT is less than sT, the search is backwards.

sp1   This is the amplitude for peaks, threshold level for threshold crossings, baseline level for maximum excursion and number of points for mode 14. 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 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/bins for slope measurements. It sets the minimum time that the data must be above/below a level for threshold measures. It sets the maximum allowed width of a peak or trough. If omitted, 0 is used. Set 0 if `width` is not required for the mode or you do not want a time constraint.

flag% This is the sum of flag values and is 0 if omitted. The only flag value defined is 1=ignore gaps in waveform data. If this value is not set, a search of a waveform channel will stop at a gap in the data.

lv2   This is the second level for modes 19 and 20. If omitted, 0 is used.

Returns A positive position in seconds or bins if the search succeeds, -1 if it fails or a negative error code.

See also:`ChanMeasure()`, `ChanValue()`, `CursorActive()`

ChanSelect()

This function is used to report on the selected/unselected state of a channel in a time or result view, and to change the selected state of a channel.

```
Func ChanSelect(chan% {,new%});
```

chan% The channel number or -1 for all visible channels. Hidden channels and non-existent channels always appear to be unselected.

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 The channel state at the time of the call, 0 for unselected, 1 for selected. If you set `chan%` to -1, the function returns the number of selected channels.

See also:`ChanHide()`, `ChanOrder()`, `ChanShow()`, `ChanWeight()`

ChanShow()

Display a channel, or a list of channels in a time, result or XY view. Turning on a channel that is on has no effect. Turning on a channel that doesn't exist has no effect.

```
Proc ChanShow(cSpc {,cSpc...});
```

cSpc   A channel specification for the channels to show.

See also:`ChanHide()`, `ChanList()`, `ChanVisible()`

ChanTitle$()

This returns or sets the channel title string in a time, result or XY view. A title string is up to 9 characters long. In an XY view, the channel titles are visible in the Key window.

Setting a duplicate channel title does not change the original channel. Setting the original channel title sets the titles of duplicates unless the duplicates have their own title. If you set the title of a duplicate to `""`, the title reverts to the title of the channel it duplicates.

```
Func ChanTitle$(chan%{,new$});
```

chan% The channel number. In XY views you can also use 0 to change the y axis title.

new$ An optional string holding the new channel title.

Returns The channel title string for the channel. If the channel does not exist, the function does nothing and returns an empty string.

See also:ChanComment$(), XYKey()

ChanUnits$()

This gets or sets the units for waveform or WaveMark channels and result or XY views. From 5.03 you can read back and set the displayed units of time view event-based channels. Changes to event-based channels are lost when the display mode changes.

```
Func ChanUnits$(chan%{,new$});
```

chan% A channel in the time or result view. In an XY, view the channel number is ignored. We suggest you use 0 to match ChanTitle$().

new$ An optional string holding the new y axis units for the channel (up to 5 characters). If the string is too long, it is truncated.

Returns It returns the old units of the designated 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:ChanScale(), ChanOffset(), ChanTitle$()

ChanValue()

In time views, this returns the value of a channel at a time. For a waveform channel it returns the waveform value, for other channel types, it returns a value in the y axis units of the channel display mode. If the channel has no y axis or is drawn in raster mode, the value is the time of the next event on the channel. In a result view, this returns the value of the result corresponding to an x axis value. You can use the [bin] notation to access result views by bin number.

```
Func ChanValue(chan%,x{,&data%{,mode%{,bsz{,trg%|edge%{,as%}}}}});
```

chan% The channel number in the time or result view.

x   In a time view, the time for which the value is needed. In a result view, this is the x axis value for which a result is needed.

data% Returned as 1 if there is data at pos, 0 if not. For example, for a waveform if there was no data within Binsize(chan%) of pos, this would be set to 0.

mode% If present, this and the following arguments sets the display mode in which to read the value from a time view. If mode% is inappropriate or absent, the current display mode is used. See DrawMode() for details. This parameter is ignored in a result view. The modes in a time view are:

   0   The current mode for the channel. Any additional arguments are ignored.
   1   Dots mode for events. Returns the event time at or after x.
   2   Lines mode, result is the same as mode 1.
   3   Waveform mode.
   4   WaveMark mode, returns waveform values for WaveMark channels.
   5   Rate mode. The binSz argument sets the width of each bin.
   6,11 Mean frequency mode, binSz sets the time period, 11 is rate per minute.
   7,12 Instantaneous frequency, returns value at next event, 12 is per minute.
   8   Raster mode, trig% sets the trigger channel, result as for mode 1.
   13   Cubic spline for waveforms and WaveMark data.
   16   Skyline mode for waveform, RealWave and RealMark channels.

Returns It returns the value or 0 if no data is found. For waveform data, if there is no data within Binsize(chan%) of the time, the value is zero.

If data% is omitted any error stops the script. Errors include: no current window, current window not a time or result view, no data at x, and x beyond the range of the x axis. If data% is present, errors set it to 0.

In instantaneous frequency mode, the value returned is for the next value when drawn in dot mode, otherwise it is the displayed value at time x.

See also:ChanData(), DrawMode()

ChanVisible()

This returns the state of a channel in a time, result or XY view 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 to report on.

Returns 1 if the channel is displayed, 0 if it is not.

See also:ChanShow(), ChanHide()

ChanWeight()

This sets the relative vertical space for a channel or a channel list. The standard space corresponds to a weight of 1. When allocating space, channels are of two types: those with a y axis and those without. Spike2 calculates the space to give each channel type assuming all channels have a weight of 1. The actual space 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()

ChanWriteWave()

This function writes real or integer data to a waveform (16-bit integer) or RealWave (32-bit floating point) channel. The time gap between array points is the Binsize() value of the channel. You can overwrite existing data and add data to the end of the channel. You cannot fill in gaps in wave channels; values written into gaps in data are ignored.

```
Func ChanWriteWave(chan%, arr[]|arr%[], sTime);
```

chan%   The waveform or RealWave channel in the current time view to write data to. This can be a duplicate channel, a disk channel or a memory channel. If you write to a duplicated channel, the original channel data is changed.

arr   A real or integer array to write to the channel. When writing real data to a waveform channel or integer data to a RealWave channel, the data is converted to match the channel format using the channel scale and offset. When writing to a waveform, output is limited to 16-bit integers in the range -32768 to 32767.

sTime   The first array point time. When overwriting, if the time does not align with existing data it is reduced by less than one sample interval to align it.

Returns The number of points processed including points skipped due to gaps in existing channel data or a negative error code, for example if the file is read-only.

The function will cause a fatal script error if used on the wrong view type, the wrong channel type or if the system runs out of memory.

See also:Binsize(), ChanData(), ChanNew(), ChanOffset(), ChanScale()

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.

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

Colour()

This function gets and/or sets item colours using the palette. XY channels are coloured using XYColour(). The colours for time and result view drawing modes can be overridden by ChanColour(). Use ColourGet() and ColourSet() in new scripts.

```
Func Colour(item% {,col%});
```

item%   This selects the item to be coloured. The available items are:

0 Channel numbers	14 Result dots	28 Cursors
1 Time background	15 Result skyline	29 Controls (unused)
2 Waveform channel	16 Result histogram	30 Grid colour
3 Events as dots	17 Result histogram fill	31 X and Y Axes
4 Events as lines	18 Marker code 00	32 XY background
5 Level events	19 Marker code 01	33 Not saving to disk
6 Marker data and code	20 Marker code 02	34 Result SEM and SD
7 Mean frequency	21 Marker code 03	35 Fitted curves
8 Inst. Frequency	22 Marker code 04	36 Cluster background
9 Raster dots	23 Marker code 05	37 WaveMark background
10 Rate outline	24 Marker code 06	38 Vertical markers
11 Rate fill	25 Marker code 07	39 XY channels
12 Result background	26 Marker code 08	40 XY fill
13 Result lines	27 TextMark text	41 XY key labels

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.

Returns The index into the colour palette of the colour of the item at the time of the call.

See also:ChanColour(), PaletteGet(), PaletteSet(), XYColour()

ColourGet()

Get the RGB colour for an item in the palette, the main colour table or the marker colour table. Added at version 7.07.

```
Func ColourGet(table%{, item%, &r, &g, &b});
```

table% -1 to select the colour palette, 0 for the main colour table, 1 for the marker table.

item%   The item number in the table selected by table%. See ColourSet().

r g b   If present, returned as the red, green and blue values in the range 0 to 1.0.

Returns If only the table number is supplied, returns the length of the table, or 0.

See also:ChanColourGet(), ColourSet(), ViewColourGet()

ColourSet()

Set the RGB colour for an item in the palette, the main colour table or the marker colour table. Added at version 7.07.

```
Func ColourSet(table%, size%);
Func ColourSet(table%, item%, r, g, b);
```

table% -1 to select the colour palette, 0 for the main colour table, 1 for the marker table.

size%   Used to set the size of the marker colour table (`table%` = 1) in the range 3 to 256. The initial, default value is 9 for marker colours 0-8. If you increase the size, the new table elements are set to black. Setting a size of 0 resets the table to the default size. Setting a size of -1 resets the table size and sets default colours.

item%   The item number in the table selected by `table%`. This is an index from 0 to 39 for the colour palette. It is one of the following for the main colour table:

0 Channel numbers	11 Rate fill	22 X and Y Axes
1 Time background	12 Result background	23 XY background
2 Waveform channel	13 Result lines	24 Not saving to disk
3 Events as dots	14 Result dots	25 Result SEM and SD
4 Events as lines	15 Result skyline	26 Fitted curves
5 Level events	16 Result histogram	27 Cluster background
6 Marker data and code	17 Result histogram fill	28 WaveMark background
7 Mean frequency	18 TextMark text	29 Vertical markers
8 Inst. Frequency	19 Cursors	30 XY channels
9 Raster dots	20 Controls (unused)	31 XY fill
10 Rate outline	21 Grid colour	32 XY key labels

For the marker colour table it is an index from 0 up to the table size -1.

r g b   If present, sets the red, green and blue values in the range 0 to 1.0.

Returns If only the table number is supplied, returns the length of the table, or 0.

See also:`ChanColourSet()`, `ColourGet()`, `ViewColourSet()`

Conditioner commands   The `Cond…` family of commands control signal conditioners. They support the CED 1902 programmable signal conditioner, the Power 1401 programmable gain options and the Axon Instruments CyberAmp. Other conditioners may be added in the future.

These commands do not select which serial port (if any) the conditioner uses or the type of conditioner supported. You choose the conditioner type when you install Spike2. You set the serial port in the Edit menu Preferences option. All these commands require a `port%` argument. This is the physical waveform input port number that the conditioner is attached to. It is not a channel number in a time view.

You can access the built-in interactive support for the conditioner from the Sampling Configuration channel parameters 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()`, `CondFilterType()`, `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.

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()

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()`

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()`,`CondFilterList()`

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:`CondGainList()`,`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:CondGain(), CondOffset(), CondOffsetLimit(), 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:Abs(), ATan(), Cosh(), Exp(), Ln(), Log(), Pow(), Sin(), 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()

Count()

In a time view, this counts events and forms the mean level of a waveform channel. It can also be used in a result view to sum bin contents between a start and end bin number.

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

chan%   The channel number in a time or result view. If the channel does not exist, -1 is returned.

start   The start time/bin. If start is greater than finish, the result is 0.

finish   The last time/bin. In a time view, if start equals finish, only items that fall exactly at the time count towards the result.

Returns   In a time view, for waveform channels it returns the mean waveform level in the time range (gaps in waveforms are ignored). If the time range falls entirely in a gap, the result is 0. For all other channels, it returns the number of events. In a result view, it returns the sum of the bins in the range.

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

Cursor()

This returns a cursor position and optionally sets a new position. If you move cursor 0 in a time view, all active cursors 1-9 will search, equivalent to CursorSearch(1).

```
Func Cursor(num% {,where});
```

num%   The cursor number to use in the range 1 to 9 (0 to 9 in a time view).

where   If present, the new cursor position. If this exceeds the x axis range, it is limited to the x axis. In a time view, the position is in seconds. In a result view it is a bin number, use XToBin() to convert an x axis value to a bin number. In an XY view the position is in x axis units.

Returns   The old cursor position (in bins in a result view) or -1 if the cursor doesn't exist.

See also:XToBin(), CursorDelete(), CursorLabel(), CursorLabelPos(), CursorNew(), CursorRenumber(), CursorSet(), CursorVisible()

CursorActive()

This function retrieves the current active cursor mode and optionally sets a new mode and search parameters. This is valid for a time view only. The function is equivalent to the vertical cursor mode dialog. Once you have set a cursor mode, you can command it to seek with CursorSearch() and tell if the search succeeded with CursorValid().

```
Func CursorActive(num%);
Func CursorActive(num%, mode%{, ch%, str$|minSt, end${, def${,
                  ampLev|n%{, hyPer{, width{, ref${, lev2}}}}}}});
```

num%   This is the cursor number, from 0 to 9.

mode%   If this argument is present, it sets the new cursor mode. Modes in *italics* cannot be used for cursor 0 and are converted to Static mode. See the documentation for the cursor mode dialog for details of each mode.

0	Static	7	Rising threshold	14	Data points
1	*Maximum value*	8	Falling threshold	*15*	*Repolarisation %*
2	*Minimum value*	9	*Steepest rising*	16	Expression
3	*Maximum excursion*	10	*Steepest falling*	17	Turning point
4	Peak find	*11*	*Steepest slope (+/-)*	*18*	*Slope%*
5	Trough find	12	Slope peak	19	Outside thresholds
6	*Threshold*	13	Slope trough	20	Inside thresholds

ch%   The time view channel to search. Use 0 for modes that do not require a channel.

str$ This is an expression that sets the start time for the search when num% is not 0. In expression mode (16), this is the expression to evaluate.

minSt   This sets the minimum step for cursor 0 in all modes except 0 and 16.

end$ This string expression sets the end limit of the search. This is ignored for cursor 0 operations when it should be an empty string.

def$ Optional. If a search fails, and this string evaluates to a valid time, the cursor is positioned at the time and the position is valid. This is ignored for cursor 0 operations when it should be an empty string.

ampLev A value or string that sets the amplitude for peaks, 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 or an empty string if ampLev is not required for the mode.

n%   The data point count for mode 14. A value of 0 is treated as 1.

hyPer   The hysteresis for threshold crossings and percent for modes 15 and 18. If omitted, 0 is used. Set it to 0 if hyPer is not used by the mode.

width   This is the width in seconds for all slope measurements. It sets the reference level measurement width in mode 15. It sets the minimum time that the data must be above/below a level for threshold measures. It sets the maximum allowed width of a peak or trough. If omitted, 0 is used. Set it to 0 if width is not used in the mode or you do not want a time constraint.

ref$ This string expression is used in mode 15 to set the time at which the 100% value is measured. The 0% value is measured at the start time.

lev2   This sets the second threshold level in modes 19 and 20. If omitted, 0 is used. You can supply this as a value or as a string to evaluate.

Returns The active cursor mode at the time of the call.

The arguments str$, end$, def$ and ref$ are strings holding expressions that evaluate to a time in seconds. They are typically of the form "Cursor(0)+1.3". They can contain any expression that would be valid in the Cursor mode dialog.

See also:CursorActiveGet(), CursorSearch(), CursorValid()

CursorActiveGet()

This gets active cursor parameters set by `CursorActive()` or the cursor mode dialog.

```
Func CursorActiveGet(num%, item% {,&val$});
```

num%   This is the cursor number from 0 to 9.

item%   This specifies the parameter value as defined for `CursorActive()` to return:

1 ch%	3 end$	5 ampLev|n%	7 width	9 ref$
2 str$	4 def$	6 hyPer	8 minSt	10 lev2

val$ This optional string variable is updated with the parameter value when item% is 2, 3, 4, 5, 9 or 10.

Returns The numerical value of items 1, 5, 6, 7, 8 and 10. Items 5 and 10 may return 0 if the item is an un-parseable string. Otherwise, it returns the active cursor mode.

See also: `CursorActive()`, `CursorSearch()`, `CursorValid()`, `MeasureX()`

CursorDelete()

Deletes a cursor. It is not an error to delete an unknown cursor (which has no effect). You cannot delete cursor 0; use `CursorVisible(0,0)` to hide cursor 0.

```
Func CursorDelete({num%});
```

num%   The cursor number to delete. If omitted, the highest numbered cursor is deleted. Use -1 to delete all cursors 1-9.

Returns The value of num% if any cursors were deleted or 0 if no cursor was deleted.

See also: `Cursor()`, `CursorNew()`, `CursorRenumber()`, `CursorSet()`

CursorExists()

Use this function to determine if a vertical cursor exists.

```
Func CursorExists(num%);
```

num%   The cursor number in the range 0-9. Cursor 0 always exists in a time view and never exists in any other view.

Returns 0 if the cursor does not exist, 1 if it does.

See also: `CursorDelete()`, `CursorNew()`, `CursorValid()`, `HCursorExists()`

CursorLabel()

This gets and optionally sets the cursor label style for the view or for a cursor. You can label cursors with a position and/or the cursor number, or with user-defined text. There are two variants: the first sets styles and labels; the second reads back user-defined labels.

```
Func CursorLabel({style%{, num%{, form$}}});
Func CursorLabel(&form$, num%);
```

style% Set 0=None, 1=Position, 2=Number, 3=Both, 4=User-defined. Omit for no change. Style 4 is used with a format string. Styles 0-3 set the styles of cursors selected by num% and the view style for new cursors if num% is -1 or omitted. Style 4 is applied to cursors set by num%; it does not set the view style.

num%   A value of -1 or omitting the argument selects all cursors and sets the view style for new cursors, 0-9 selects one cursor.

form$ A user-defined label string with replaceable fields %p, %n and %v(chan) for position, number and channel value; chan is 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 A cursor style before any change as 0-4 if a single cursor is selected, or the view cursor style as 0-3. If `style%` is omitted or not 0-3, the current view cursor style is not changed.

See also:`Cursor()`, `CursorLabelPos()`, `CursorNew()`, `CursorRenumber()`

CursorLabelPos()

This lets you set and read the position of the cursor label.

```
Func CursorLabelPos(num% {,pos});
```

`num%`   The cursor number. Setting a silly number 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.

See also:`Cursor()`, `CursorLabel()`, `CursorNew()`, `CursorRenumber()`

CursorNew()

This command adds a new cursor to the view at the designated position. You cannot use this to create cursor 0 in a time view as this cursor always exists. To show cursor 0 use `CursorVisible(0,1)`. A new cursor is created in Static mode (not active).

```
Func CursorNew({where{, num%}});
```

`where`   The cursor position. In a time view it is a time in seconds, in a result view, it is the bin number. Use `XToBin()` to convert x axis units to bin numbers. 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()`, `CursorSet()`, `CursorVisible()`, `XToBin()`

CursorOpen()

This command reports on the open state or opens the cursor values and regions dialogs for the current Time or Result view. Dialogs open in the last used position unless not usefully on a monitor, when they open centred on the application window. If the dialog is open you can use this command to get the dialog handle and change settings. Use `FileClose()` to close an opened dialog.

```
Func CursorOpen({opt%{, mode%{, xZero%{, yZero%|type%}}}});
```

`opt%`   Set 0 to open the values dialog and 1 to open the regions dialog. Omit or set -1 as the only argument to report on the open state of the cursor dialogs.

`mode%`   Set 1 to open the dialog and show it, 0 or omitted to open and hide it.

`xZero%`   This sets the state of the Time Zero or Zero Region check boxes and the associated cursor column selection. Set -2 for unchecked, -1 or omit for no change or set 0 to 9 for the values or 0 to 8 for the regions dialog to check the box and select a column. 0 selects the first column, 1 the second, and so on.

`yZero%`   This sets the state of the Y Zero check box and associated column selection for the value dialog. Set -2 for unchecked, -1 or omit for no change and 0-9 to check the box and select a column. 0 selects the first column, 1 the second, and so on.

`type%`   This sets the measurement type for the regions dialog. Set 1-19 to set a measurement as for `ChanMeasure()`. Omit or set -1 or 0 for no change.

Returns If `opt%` is -1 or omitted the return value is the sum of 1 if the values dialog is open and 2 if the regions dialog is open. Otherwise the return value is the handle of the opened dialog, or 0 if the dialog failed to open for some reason.

See also:`ChanMeasure()`, `ChanValue()`, `FileClose()`, `Window()`

CursorRenumber()

This command renumbers the cursors from left to right in the view. It has no effect on cursor 0. Active cursor and label information stays with the cursors, not the number.

```
Func CursorRenumber();
```

Returns The number of cursors that were renumbered (cursor 0 is not counted).

See also:`Cursor()`, `CursorActive()`, `CursorDelete()`, `CursorLabel()`, `CursorLabelPos()`, `CursorNew()`, `CursorSet()`

CursorSearch()

This function causes active cursors in a time view to search according to the current cursor mode. You can cause all cursors to search, or a restricted range of cursor numbers. Moving cursor 0 with `Cursor(0, new)` also causes all cursors 1-9 to search.

```
Func CursorSearch(num% {,stop%});
```

num%   This is the first cursor number to run the search defined by the active cursor mode. Set this to 0 to cause cursor 0 to search forwards and to -1 for cursor 0 to search backwards. `CursorSearch(0)` and `CursorSearch(-1)` are equivalent to the `Ctrl+Shift+Right` and `Ctrl+Shift+Left` key combinations.

stop%   This optional argument sets number of the last cursor to try to reposition. If you omit this argument, all cursors from num% upward will search according to their active mode. To reposition a single cursor set stop% the same as num%.

Returns The time that cursor num% moved to or -1 if didn't move. You can also use `CursorValid()` to test if searches have succeeded.

See also:`Cursor()`, `CursorActive`, `CursorActiveGet()`, `CursorNew()`, `CursorValid()`, `MeasureChan()`, `MeasureX()`

CursorSet()

This sets the number of vertical cursors in addition to cursor 0. It deletes cursors 1-9, then positions num% cursors equally spaced in the view, numbered in order from left to right. Finally, if any cursors positions are given, they are applied. The cursor labelling style is not changed. This destroys all active cursor information for the deleted cursors.

```
Proc CursorSet(num% {,where1 {,where2 {,where3 {,where4...}}}});
```

num%   The number of cursors to display in the range 0 to 9. It is a run-time error to ask for more than 9 or less than 0 cursors. If num% is 0, cursor 0 is hidden.

whereN Optional positions of cursor N (1 to 9). Positions that are out of range are set to the nearest valid position. In a time view, positions are in seconds. In a result view, they are the bin number; use `XToBin()` to convert x axis units to bin numbers. In an XY view the position is in x axis units. You cannot change the position of cursor 0 in a time view with this command.

Examples
```
CursorSet(0);        'Delete 1-9, hide cursor 0
CursorSet(2,20,30);  'Delete 1-9, cursor 1 at 20, cursor 2 at 30
```

See also:`BinToX()`, `Cursor()`, `CursorNew()`, `CursorRenumber()`, `CursorVisible()`, `XToBin()`

CursorValid()

Use this function to test if the last search of a cursor in a time view succeeded. Cursor positions are valid if a search succeeds or if the cursor is positioned manually or by a script command. The position of a newly created cursor is valid.

```
Func CursorValid(num%);
```

num%   The cursor number to test for a valid search result in the range 0-9.

Returns The result is 1 if the position of the nominated cursor is valid or 0 if it is invalid or the cursor does not exist.

See also:`CursorActive`, `CursorActiveGet()`, `CursorNew()`, `CursorSearch()`, `CursorVisible()`, `MeasureChan()`, `MeasureX()`

CursorVisible()

Vertical cursors can be hidden without deleting them. Interactively you can hide cursor 0, but from a script, you can show and hide any vertical cursor. Cursors are always made visible by the `Ctrl+n` key combination.

```
Func CursorVisible(num% {,show%});
```

num%   The cursor number in the range 0-9 or -1 for all vertical cursors.

show%   If present set this to 0 to hide the cursor and non-zero to show it.

Returns The state of the cursor at the time of the call (0=hidden, 1=visible) or -1 if the cursor does not exist. If num% is -1, the result is the number of vertical cursors.

See also:`CursorExists()`, `CursorNew()`, `CursorSearch()`, `CursorValid()`

Date$()

This function returns a string holding the date. Use `TimeDate()` to get the date as numbers. For this description, we assume that today's date is Wednesday 1 April 1998, the system language is English and the system date separator is "/". Default argument values are shown **bold**.

```
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. The options are:

    1   Show day of week: "`Wednesday`".
    2   Show the number of the day in the month with leading zeros: "`01`".
    4   Show the day without leading zeros: "`1`". This overrides option 2.
    8   Show abbreviated day of week: "`Wed`".
    16 Show weekday name first, regardless of the order% field.

Use 0 for no day field. Add the numbers for multiple options. For example, to return "`Wed 01`", use 11 (1+2+8) as the dayF% argument.

If you add 8 or 16, 1 is added automatically. If you request both the weekday 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: "`April`".
    2   Show number of month: "`04`"
    3   Show an abbreviated name of month: "`Apr`"
    4   Show number of month with no leading zeros: "`4`"

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.

    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, Spike2 supplies a separator based on 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(),FileDate$(),TimeDate(),Time$()

Debug()

This command either opens the debug window so you can step through your script, set breakpoints and display and edit variables or it can be used to stop the user entering the debugger with the Esc key.

```
Proc Debug({msg$}|{Esc%});
```

msg$ When the command is used with no arguments, or with a string argument, the script stops as though the Esc key had been pressed and enters the debugger. If the debugging toolbar was hidden, it becomes visible. If the msg$ string is present, the string is displayed in the bar at the top of the script window.

Esc%   When the command is used with an integer argument, it enables and disables the ability of the user to break out of a running script. If Esc% is 0, the user cannot break out of a script into the debugger with the Esc key and must wait for it to finish. If Esc% is 1, the user can break out. Spike2 enables the Esc key each time a script starts, so make this the very first instruction of your script if you want to be certain that the user cannot break out.

This 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 before using this option. Once set, you cannot stop a looping script except by forcing a fatal error. Make sure you save your script before setting this option.

See also:Eval()

DebugHeap()

This script command is provided for use by CED engineers to help to debug system problems. It reports on the state of the application heap, used to dynamically allocate memory. The heap is a list of memory sections. Each section is described by its start address, its size, and if it is in use by the application or is free (available for use). When the application wants more memory, it asks the heap for it. If there is no suitable memory in the heap, the heap requests more memory from the system and uses this to create more heap sections. The command has the following variants:

Test heap integrity   When called with no arguments, the command tests the integrity of the heap (all command variants do this first). The return values indicate problems in the heap. The error return values apply to all calls to the DebugHeap() function.

```
Func DebugHeap();
```

Returns The function returns 0 if no error is detected, otherwise a negative error code:

-1 Initial header information is bad or cannot be found.
-2 Bad node or the heap is damaged.
-3 A pointer into the heap is invalid.
-4 The heap has not been initialised (should never occur)
-5 Unknown error

Release unused memory   When called with a single argument of -1 (or any value less than 0), the heap will return unused memory back to the operating system. However, I have never seen this make any difference to the data held by the heap.

```
Func DebugHeap(-1);
```

Returns The same values as the call with no arguments.

Set unused heap memory to   When called with a single argument that is a positive number, all unused memory that is
known value   owned by the heap is set to this value.

```
Func DebugHeap(fill%);
```

fill%   All unused heap bytes are set to the low byte of this value. This can sometimes be useful when you suspect that unused memory is being used by the program.

Returns The same values as the call with no arguments.

Get heap information   The heap is a list of used an unused sections of memory. This call gets information on the number of used an unused sections and the total size of the used sections and the total heap size.

```
Func DebugHeap(info%[]{, stats%[]});
```

info%   An integer array of at least 4 elements. The first four array elements are returned holding:

0 The total number of memory sections in the heap.
1 The number of used memory sections in the heap.
2 The total size of memory controlled by the heap.
3 The total size of memory that is in used and controlled by the heap.

stats%   This optional argument is an integer array of at least 32 elements. The elements are returns with heap information. The nth element is returned holding the number of heap sections of a size between $2n$ and $2n+1-1$ bytes. The very first element, which should hold the count of sections that are 1 byte in size actually holds the count of sections that are 0 or 1 byte long.

Returns The same values as the call with no arguments.

Get heap entries   The last call variant returns the entire heap information, and can optionally return the first few bytes of data held in the heap.

```
Func DebugHeap(walk%[][]);
```

walk%   This is an integer matrix with at least 3 columns. Ideally it would have at least as many rows as there are memory sections (both used and free) in the heap. If there are more than 3 columns, the additional columns are returned holding the heap data. Each row of the matrix returns information for one section. The columns hold:

0 0 if the section is unused, 1 if it is used.
1 The start address of the section.
2 The size of the section, in bytes.
3 If present, returns the first 4 bytes of data in this section.
4 If present, returns the next 4 bytes of data in this section.
n Returns further data (if it exists) in the section.

Returns If there is no error, the function returns the number of sections in the heap. This can be more than the number of rows in the matrix. If there is an error, the function returns a negative value as described for the call with no arguments.

DebugList()

This command is used for debugging problems in the system. It writes information to the Log view about the internal list of "objects" used to implement the script language. It can also be used to diagnose if a script is spending a lot of time in a built-in function.

```
Proc DebugList(list% {, opt%});
```

list% Positive values determine what to list. 0 lists a summary of the options, 1 lists fixed objects (constants and operators), 2 lists permanent objects (constants, operators and built in commands) and 3 lists built-in commands. Values greater than 3 list information for the object with that number. Negative values control the accumulation of timing information for built-in functions. -3 disables timing (default), -2 enables timing, -1 resets accumulated times and call counts. See the online help for an example.

opt% An optional argument (default 0) to list more information and is the sum of: 1= index number, 2= type, 4= timing information.

See also:Debug(), Eval(), DebugOpts()

DebugOpts()

This command is used for debugging problems in the system. It controls internal options used for debugging at the system level.

```
Func DebugOpts(opt% {,val%});
```

opt% This selects the option to return (and optionally to change). A value of 0 prints a synopsis of available options to the Log view and the current value of each option. Values greater than 0 return the value of that option, and print the option information to the Log view. At the time of writing, only option 1, dump compiled script to the file default.cod is implemented.

Val% If present, this sets the new value of the option.

See also:Debug(), Eval(), DebugList()

DelStr$()

This function removes a sub-string 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 remove. If this would extend beyond the end of the string, the remainder of the string is removed.

Returns The original string with the indicated section deleted.

See also:Asc(), Chr$(), InStr(), LCase$(), Left$(), Len(), Mid$(), Print$(), ReadStr(), Right$(), Str$(), Trim(), UCase$(), Val()

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. You are allowed up to 1000 items in a dialog.

The dialog has a title of your choosing at the top and OK and Cancel plus user-defined buttons at the bottom. When the dialog is used, pressing the Enter key is equivalent to clicking on OK.
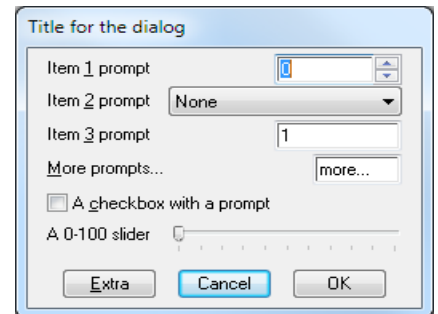
This form of dialog is very easy to program; there is no need to specify any position or size information, the system works it all out. In more complex cases, you specify the position and width of the dialog and the positions of all the dialog items.

Dialog operations You can set specific dialog items for integer and real numbers, text strings, a time or x axis value and add a group box. You can define script functions that are called in response to button presses, user changes to the dialog and an idle-time function that is called repeatedly while the dialog waite for user actions. All these functions can enable, disable, hide, show and modify dialog items. You can add tooltips for all the prompts in a dialog to give users a more detailed explanation of a field.

Version 7 extensions From version 7.01 onward, you can define script functions that are called when the mouse pointer moves and when the left-hand mouse button is clicked. See `DlgMouse()`.

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. The maximum y position is 40. 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|Tip", 1);    'Waveform channel list
DlgReal   (3, "Item &3 prompt", 1.0, 5.0)  'real, range 1.0-5.0
DlgString (4, "&More prompts...|Tip", 6);  'string, any characters
DlgCheck  (5, "A &checkbox with a prompt");'a checkbox item
DlgButton (2, "&Extra||Tooltip");          'button 2, tooltip

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. 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.
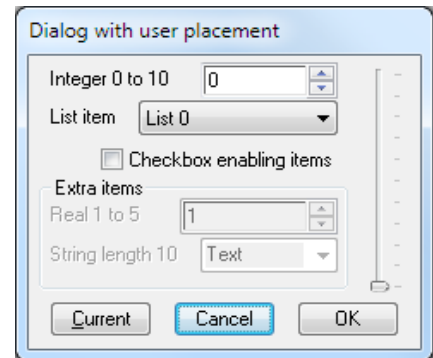
With `DlgAllow()` we have set `Func Change%(item%)` to be called whenever the user changes a selection or checkbox or when an editable field loses the input focus. The `item%` argument is set to the item number that changed or to 0 if the dialog is appearing for the first time. We are interested in item 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 and can be activated with the `C` and `F1` keys and has a tooltip. The function displays a message box that lists the current values of items in the dialog.

```
var ok%, item1%, item2%, item3%, item4, item5$:= "Text", gp%;
DlgCreate("Dialog with user placement",0,0,40,7.5);
DlgInteger(1,"Integer 0 to 10",0,10,0,1,1);     'Int with spinner
DlgList(2,"List item","List 0|List 1|List 2|List 3", 4, 0, 2);
DlgCheck(3, "Checkbox enabling items",0,3);     'checkbox item
DlgReal(4, "Real 1 to 5",1.0,5.0,0,4.5,0.5);    'Real with spinner
DlgString(5, "String length 10",10,"",0,5.5,    'String item with
             "String 1|String 2|String 3");     ' drop-down list
DlgButton(2,"&Current|Ox70|Tooltip", Current%);'button F1+function
DlgAllow(0x3ff, 0, Change%);  'Allow all, no idle, change function
gp% := DlgGroup("Extra items",1,3.8,-1,2.9);    'Group box
ok% := DlgShow(item1%,item2%,item3%,item4,item5$);
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$;
v1% := DlgValue(1); v2% := DlgValue(2); 'Retrieve values
v3% := DlgValue(3); v4  := DlgValue(4); v5$ := 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()`, `DlgGetPos()`, `DlgGroup()`, `DlgInteger()`, `DlgLabel()`, `DlgList()`, `DlgReal()`, `DlgShow()`, `DlgString()`, `DlgText()`, `DlgValue()`, `DlgVisible()`, `DlgXValue()`, `ToolbarSet()`

DlgAllow()

Call this after `DlgCreate()` and before `DlgShow()` ends 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()` cannot be used during `DlgShow()` before version 6.11.

```
Proc DlgAllow(allow% {,func id%(){, func ch%()}});
```

allow% A number that specifies the actions that the user can and cannot take while interacting with Spike2. 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, Spike2 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: `DlgCreate()`, `DlgEnable()`, `DlgShow()`, `DlgValue()`, `DlgVisible()`, `Interact()`, `ToolbarSet()`

DlgButton()

Dialogs created by `DlgCreate()` have Cancel and OK buttons; this function adds, deletes and changes buttons. 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()`. There are two variants:

```
Func DlgButton(but%, text${, func ff% {, x, y}});        ' Set a button
Func DlgButton();                  ' Get the last used button while the dialog is active
```

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.

A button with a key code 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 and use the key code without clicking in the dialog.

ff%()   This integer function with no arguments 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 Cancel. The dialog closes, `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.

Returns 0 unless called with 0 arguments when it returns the last used button number.

See also: `DlgEnable()`, `DlgShow()`, `DlgValue()`, `DlgVisible()`, `ToolbarSet()`

DlgChan()

This function defines a dialog entry that lists channels that meet a specification for time, result and XY views. 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 time, result or XY view, the list is empty.

```
Proc DlgChan(item%, text$|wide, mask%|list%[]{, x{, y}});
```

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 used to select a channel. If `wide` is omitted the number entry box has a default width of the longest channel name in the list or 12, whichever is the smaller.

mask%   This determines the channels to display. Select channel types by adding the following codes, which are given as decimal and hexadecimal.

1	0x1	Waveform or result view channel
2	0x2	Event+ and Event- channels
4	0x4	Event +- channels (level data)
8	0x8	Marker channels
16	0x10	WaveMark data
32	0x20	TextMark data
64	0x40	RealMark data
128	0x80	Unused/deleted disk channels
256	0x100	Deleted channels on disk
512	0x200	Real wave channel

If none of the above values is used or this is an XY view, the list includes all channels. Add the following codes to exclude channels from the list:

1024	0x400	Exclude visible channels
2048	0x800	Exclude hidden channels
4096	0x1000	Exclude time view disk channels but not their duplicates
8192	0x2000	Exclude memory channels but not their duplicates
16384	0x4000	Exclude duplicated channels
32768	0x8000	Exclude selected channels
65536	0x10000	Exclude non-selected channels
2097152	0x200000	Exclude virtual channels but not their duplicates

Finally, the following add special entries to the list and control the display:

131072	0x20000	Add `None` as an entry in the list, returns 0
262144	0x40000	Add `All channels` as an entry in the list, returns -1

524288	0x80000	Add `All visible channels` as an entry, returns -2
1048576	0x100000	Add `Selected` as an entry in the list, returns -3
4194304	0x400000	No channel type or shorten long XY channel titles

list%
As an alternative to a mask, you can pass in a channel list (as constructed by `ChanList()`). This must be an array of channel numbers, with the first element of the array holding the number of channels in the list. From version 6.03, you can add special entries into the first element. For example, if you have 5 channels following and you want to add `None` as an option, use `list%[0]:=0x20005`.

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 in dialog units.

y
If omitted or zero, this takes the value of `item%`. It is the position of the bottom of the channel selection box in dialog units.

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:`DlgCheck()`,`DlgCreate()`,`DlgInteger()`,`DlgLabel()`,`DlgList()`,
       `DlgReal()`,`DlgShow()`,`DlgString()`,`DlgText()`,`DlgXValue()`

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 in the range 1 to the number of items.

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 check box in dialog units. If omitted, x is set to 2 and y to `item%`. When used without these fields, this behaves exactly like the simple dialog functions, and can be mixed with them.

The associated `DlgShow()` variable should be an integer. It sets the initial state (0 for unchecked, not 0 for checked) and returns the result as 0 (unchecked) or 1 (checked). This item does not have a prompt. If you use `DlgValue()`, a string value refers to the text and a numeric value refers to the check box state.

See also:`DlgChan()`,`DlgCreate()`,`DlgInteger()`,`DlgLabel()`,`DlgList()`,
       `DlgReal()`,`DlgShow()`,`DlgString()`,`DlgText()`,`DlgXValue()`

DlgCreate()

This function starts the definition of a dialog and clears any `DlgAllow()` settings. It also kills off any previous dialog that might be partially defined.

```
Func DlgCreate(title${,x{,y{,wide{,high{,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 as a percentage 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, Spike2 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, Spike2 works it out for itself, based on the dialog contents.

help   A string or numeric identifier that identifies the help page to be displayed if the user requests help when the dialog is displayed. Use -1 for no defined help page.

scr%   Screen selector, see `Window()` for details. Added at version 6.03.

rel%   Screen selector mode, see `Window()` for details. Added at version 6.03.

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.

See also: `DlgAllow()`, `DlgShow()`, `Window()`

DlgEnable()

Use this only from a dialog call-back function to enable or disable dialog items. With one argument, it returns the enabled state of an item; with two or more arguments, it sets the enabled state of one or more items. Prompts and spin controls associated with the item are also enabled or disabled. `DlgEnable()` was new in version 5.

```
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()`, `DlgCreate()`, `DlgShow()`, `DlgValue()`, `DlgVisible()`

DlgGetPos()

Use this only from a dialog call-back function to get the dialog position. `DlgGetPos()` was added at version 6.06.

```
Func DlgGetPos(&x, &y{, scr%{, rel%}});
```

x,y   Returned holding the position of the top left hand corner of the dialog relative to the rectangle defined by `scr%` and `rel%`.

scr%   Screen selector, see `Window()` for details.

rel%   Screen selector mode, see `Window()` for details.

Returns   1 if x and y set or -1 if the rectangle set by `scr%` and `rel%` is of zero size.

See also: `DlgAllow()`, `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 5.

```
Func DlgGroup(text$, x, y, width, height);
```

text$ The text to display at the top left of the group box. Any tooltip text is ignored.

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$` arguments are not used. The `sp` and `li$` arguments were new in version 5.

```
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:`DlgChan()`, `DlgCheck()`, `DlgReal()`, `DlgShow()`, `DlgString()`

DlgLabel()

This function sets an item with no editable part, used as a label. For simple dialogs, the `wide`, `x` and `y` arguments are not used. `DlgText()` adds text without item numbers.

```
Proc DlgLabel(item%, text${, x{, y}});
```

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.

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 in dialog units.

y   If omitted or zero, this takes the value of `item%`. It is the position of the bottom of the text in the dialog in dialog units.

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:`DlgCreate()`, `DlgList()`, `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$|wide, list$|list$[]{, n%{, x{, y}}});
```

`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. The list box width is the smaller of the longest string in the list or 20.

`wide`   This is an alternative to the prompt. It sets the width of the box in which the user selects an item. If you need a prompt, you can use `DlgText()`.

`list$`   A string of list items separated by a "|" character or an array of strings, one per item. Long strings are truncated. The "|" method was new in version 3.

`n%`   The number of entries to display. If this is omitted, or if it is larger than the number of entries provided, then all the entries are displayed.

`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`   The bottom of the selection box position. If omitted, the value of `item%` is used.

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 using an array:

```
var ok%,which%:=0;                'string list, test for OK, result
DlgCreate("List example");        'Start the dialog
var list$[3];                     'these strings are the choices
list$[0] := "one"; list$[1] := "two"; list$[2] := "three";
DlgList(1,"Choose", list$[]);     'Add the list to the dialog
ok% := DlgShow(which%);           'Display dialog, wait for user
```

From version 3, you can replace the third, fourth and fifth lines with:

```
DlgList(1,"Choose","one|two|three"); 'version 3 onwards
```

See also:`DlgChan()`, `DlgCheck()`, `DlgCreate()`, `DlgInteger()`, `DlgLabel()`, `DlgReal()`, `DlgShow()`, `DlgString()`, `DlgText()`, `DlgXValue()`

| **DlgMouse()** |

Use after `DlgCreate()` and before `DlgShow()`. There are two variants of the command. The first sets the initial mouse pointer position when the dialog opens; this is very useful if you are using multiple screens.

```
Proc DlgMouse(item%);
```

`item%`   The dialog item where the mouse pointer should move when the dialog opens. This is either the number you set for the dialog item, or the number returned by `DlgText()` or `DlgGroup()`, or `-button`, where `button` is the button number.

The second variant gives you access to the mouse position and left button clicks in Time, Result and XY views when the mouse is over a data channel while a dialog is active. See `ToolbarMouse()` for details, as the command is identical. The arguments are:

```
Proc DlgMouse(vh%, ch%, mask%, want%, Down%{, Up%{, Move%}});
```

See also:`DlgButton()`, `DlgGroup()`, `DlgShow()`, `DlgText()`, `ToolbarMouse()`

| **DlgReal()** |

This function defines a dialog entry that edits a real number. For simple dialogs, the `wide`, `x` and `y` arguments are not used. The `sp` and `li$` arguments were new in version 5.

```
Proc DlgReal(item%, text$|wide, lo, hi{, x{, y{,sp|li${,pre%}}}});
```

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"`.

pre%   If present, this sets the number of significant figures to use to represent the number in the range 6 (the default) to 15.

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: `DlgCreate()`, `DlgInteger()`, `DlgShow()`, `DlgString()`, `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 dialog is cancelled. Once the dialog has closed, 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[] ...);
```

Returns 0 if the user clicked Cancel or an associated mouse up or button function returned a negative value, or the button number clicked to close it (OK=1), or a number greater than any button if a mouse up function returned 0.

For each dialog item with an item number (including `DlgLabel()`), you must provide a variable of a suitable type to hold the result. It is an error to use the wrong variable type, except an integer field can have a real or an integer variable.

The variables also set the initial values. If an initial value is out of range, 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 simple variables, you can pass arrays. An array with `n` elements matches `n` items in the dialog. The array type must match the items.

See also: `DlgCreate()`, `DlgValue()`, `DlgVisible()`

DlgSlider()

This command adds a slider control that sets an integer value between two user-defined limits. The slider is horizontally orientated for simple use but can be vertically orientated.

```
Proc DlgSlider(item%, text$|wide, lb, rt{,tick{,flags%{,x{,y}}}});
```

item% This sets the item number in the dialog in the range 1 to the number of items.

text$ A left-justified prompt to display, optionally followed by a vertical bar and tooltip text. If `text$` is used, the slider will be horizontal and fill the space from the end of the prompt to the right hand side of the dialog.

wide   This is an alternative to the prompt. If positive, the slider is horizontal and the value sets the slider width in dialog units; a zero value uses the entire dialog

width. A negative value sets the height of a vertical slider and you will need to provide the x and y values to complete the positioning.

lb, rt   Sets the values corresponding to the left/bottom and right/top end of the slider. lb can be greater than rt but must not be the same.

tick   Optional, defaults to 0. Enables ticks if not zero, values greater than 0 set the tick spacing and negative values set tick auto-scaling in a 1, 2, 5 sequence.

flags%   This sets slider options and is the sum of: 1=tooltip value readout during drag, 2=change notifications during drag operation.

x   If omitted or zero, the slider is right justified in the dialog box, otherwise this sets the position of the left end of the slider in dialog units.

y   If omitted or zero, this takes the value of item%. It is the position of the top of the slider in dialog units.

The variable passed into DlgShow() as argument number item% should be a real. The slider starts with the value of the variable if it is in the range. Otherwise, it is limited to the nearer end of the range. Note that the range of values that a slider can return is quantised by the pixel positions that the slider can occupy.

Tooltips If you enable the tooltip value readout option in flags%, the slider value appears in any tip when the mouse is over the slider. If you have set a tip with the text$ argument, this tip appears when the mouse is over the prompt.

Change function If you have used DlgAllow() to set a change function, you can choose if this is called every time the slider position changes during a drag operation, or only when the drag operation ends.

See also:DlgCreate(), DlgInteger(), DlgShow()

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 was added 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:DlgLabel(), DlgShow(), DlgText(), Input$()

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()`. There was no returned value in Spike2 version 4.

```
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: `DlgLabel()`, `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, spinner increment, item prompt or button text.

```
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. Add 2000 to set the step value of a spin control linked to a numeric field.

val   This optional argument holds the new item value. Use val on numeric fields or to set a checkbox or an item number in a list. Use val$ to set a prompt, button label or 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: `DlgCreate()`, `DlgEnable()`, `DlgShow()`, `DlgVisible()`

DlgVisible()

Used only 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 was new in version 5.

```
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 time, result 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 time view, or an x axis value for other views. This command was new in version 5.

```
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 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 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:DlgCreate(), DlgInteger(), DlgReal(), DlgShow()

Draw()

This optionally positions the current view and allows invalid regions to update. Draw() with no arguments on a view that is up-to-date should make no change. The view is not brought to the front. Use Yield() to get a text view up to data.

```
Proc Draw({from {, size}});
```

from   The left hand edge of the window. For a time window, this is in seconds. For a result view, this is in bins. For an XY view, it is in x axis units. For a text view, this sets the top line to display in the view.

size   The width of the window in the same units as from. A negative size is ignored. This argument must be omitted for a text view.

With two arguments, the width is set (unless it is unchanged) and then it is drawn. With one argument, the view scrolls by an integral number of pixels such that from is in the first pixel. The following may not move the display if a pixel is more than a second wide:

```
Draw(XLow()+1.0); 'This may scroll by less than 1 second
```

Time views run from time 0 to the maximum time in the view. Result views have a fixed number of bins, set when they are created. XY view axes can be any positive length.

See also:DrawAll(), XRange(), XLow(), XHigh(), Maxtime(), Yield()

DrawAll()

This routine updates all views with invalid regions. Nowadays, calling Yield() is a better solution as this also allows the system time to clean up unused resources.

```
Proc DrawAll();
```

See also:Draw(), Yield()

DrawMode()	This sets and reads the channel display mode in a time or result view. You can set the display mode for hidden channels. The first command variant returns information, the second is for time views, the third is for result views and the fourth is for sonograms.

```
Func DrawMode(chan%{,mode%});
Func DrawMode(cSpc,mode%{,dotSz%|bSz|flags%{,trig%|edge%,{as%}}});
Func DrawMode(cSpc, mode%{, dotSz% {, opt%|err%{, sort}}});
Func DrawMode(cSpc, 9{, fftSz%, wnd%, top, range, xInc%, skip%});
```

chan%   The channel number used to read back information with a negative mode%.

cSpc   A channel specification or -1 for all, -2 for visible or-3 for selected channels.
        Setting a draw mode for a bad channel number has no effect.

mode%   If mode% is omitted, the function returns the mode. If negative, see below for
        return values. Positive values set the display mode. If an inappropriate mode is
        requested, no change is made. Some modes require additional parameters (for
        example a bin size). If they are omitted, the last known value is used.

        The mode values for setting modes in a time view are:
        0   The standard drawing mode for the channel. Marker channels display the
            first marker code in the standard format for the channel.
        1   Dots mode for events or a waveform. dotSz% argument can be used.
        2   Lines mode. Shows event data as vertical ticks on a horizontal line. If
            dotSz% is present and zero, the horizontal line is not drawn.
        3   Waveform mode. Draws straight lines between waveform, RealMark and
            WaveMark points. No marker codes are displayed.
        4   WaveMark mode. Only use this is you wish to see WaveMark codes as it
            takes longer to draw than Waveform mode.
        5   Rate mode. bSz sets the width of each bin in seconds.
        6   Mean frequency mode. bSz sets the time period.
        7   Instantaneous frequency mode. dotSz% and as% can be used.
        8   Raster mode. trig% sets the trigger channel, dotSz% is used.
        9   Sonogram mode. In this mode the fftSz%, wnd%, top, range, xInc% and
            skip% arguments are allowed (or can be omitted from the right).
            fftSz% The block size for the Fourier Transform used to generate the
                    sonogram. Allowed values are 16, 32, 64, 256, 512, 1024, 2048
                    and 4096. Intermediate values set the next lower allowed value.
            wnd%   The window applied to the data. 0 = no window, 1 = Hanning, 2 =
                    Hamming. Values 3 to 9 set Kaiser windows with -30 dB to -90 dB
                    sideband ripple in steps of 10 dB.
            top   The signal dB value relative to 1 bit at the ADC input to show as
                    the maximum density output. Signal values above top dB are
                    shown in maximum intensity. Values 0.0 to 100.0 are allowed.
            range   The range in dB of output data below top that is mapped into the
                    grey scale. Output below (top-range) is shown as minimum
                    intensity. Values greater than 0.0 and up to 100.0 are allowed.
            xInc%   The number of pixels (1 to 100) to step across the screen before
                    calculating the next set of sonogram values. This is normally 1.
            skip%   Normally 0. Set 1 to use one data block for each vertical strip of
                    sonogram to reduce the calculation time for large files (but only
                    shows samples of the sonogram). Values 0 and 1 are allowed.
        10 WaveMark overdraw mode.
        11 Same as mode 6, but display rate per minute rather than per second.
        12 Same as mode 7, but display rate per minute rather than per second.
        13 Cubic spline mode for waveform, WaveMark and RealMark channels.
        14 Text mode for TextMark channels. dotSz% is used.
        15 State mode for marker channels. flags% is used.
        16 Skyline mode for waveform, RealWave and RealMark channels.

For a result view channel, the modes are listed below.

0   The standard drawing mode for the result view.
1   Draw as a histogram. `err%` can be used.
2   Draw as a line. `err%` can be used.
3   Draw as dots. `err%` and `dotSz%` can be used.
4   Draw as a skyline. `err%` can be used.
8   Draw raster as lines, `dotSz%` sets the line length, `opt%` is used.
9   Draw raster as dots, `dotSz%` sets the dot size, `opt%` is used.
13 Cubic spline mode. `err%` can be used.

`dotSz%` Sets the dot or tick size to use in units of the data pen width set in the Edit menu Preferences Display tab. 0 is the smallest size available, which displays as 1 pixel. The maximum size is 10. Use -1 for no size change.

`bSz`   Sets rate histogram bin width and the mean frequency smoothing period.

`flags%` For markers in State mode, this sets the additional information to display. 0=no extra information. Add 1 to draw the state number, 2 to display TextMark text.

`trig%` This is the trigger channel for time view raster displays. We assume that you do not want to display a level event channel in raster mode.

`edge%` Sets the edges of level event data to use for mean and instantaneous frequency and rate modes. 0 = both edges (default), 1 = rising edges and 2 = falling edges.

`as%`   Instantaneous frequency draw mode. 0 or omitted =default mode (dots), 1=dots, 2=line, 3=skyline.

`opt%`   Used in result view raster channels. The default is 0. Add 1 for horizontal line in raster line mode. Add 2 for y axis as time of sweep or variable value in raster modes. Add 16 to show symbols. For backwards compatibility, if `sort%` is omitted, add 4 to select sort value 1 and 8 to select sort value 2. These values (4 and 8) are not returned when `mode%` is -12, use -15 to get the sort value.

`err%`   Result view error drawing style: 0=none (default), 1=1 SEM, 2=2 SEM, 3=SD.

`sort%` Result view raster sort mode. 0=time (default), 1-4 = use sort variables 1 to 4.

Returns If a single channel is set it returns the previous `mode%`. For multiple channels or an invalid call, it returns -1. Negative `mode%` values return drawing parameters:

-1 Reserved	-4 `trig%`	-7 `wnd%`	-10 `xInc%`	-13 `err%` -16 `as%`
-2 `dotSz%`	-5 `edge%`	-8 `top`	-11 `skip%`	-14 `sort%`
-3 `binSz`	-6 `fftSz%`	-9 `range`	-12 `opt%`	-15 `flags%`

See also: `ChanIndex()`, `Draw()`, `MarkShow()`, `SetEvtCrl()`, `SetPSTH()`, `ViewStandard()`, `XYDrawMode()`

Dup()

This gets the view handle of a duplicate of the current view or the number of duplicates. Duplicated views are numbered from 1 (1 is the original). If a duplicate is deleted, higher numbered duplicates are renumbered. See `WindowDuplicate()` for more information.

```
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, it returns the duplicate view handle, or 0 if there is no duplicate. If `num%` is 0 or omitted, it returns the number of duplicates.

See also: `App()`, `View()`, `WindowDuplicate()`

DupChan()

This returns the number of duplicates of a channel in the current time or result view, or the channel number of the nth duplicate, or it identifies the channel on which a duplicate channel is based, or it identifies which duplicate of an original channel this is.

```
Func DupChan(chan%{, num%});
```

chan%   A channel number in the current view. The channel need not exist. If this channel is a duplicate, the command behaves as though you passed the original channel on which the duplicates are based.

num%   If greater than 0, this is the index of a duplicate channel in the range 1 to the number of duplicates. Use 0 to return the original channel that was duplicated. Use -1 as an argument (or omit num%), to return the number of duplicates of the channel. Use –2 as an argument to return which duplicate a channel is: 0 means not a duplicate, 1 upwards is the duplicate index.

Returns If num% is greater than 0, it returns the duplicate channel number, or 0 if there is no duplicate. If num% is 0, it returns the channel number that chan% was duplicated from or chan% if it is the original or does not exist. If num% is –1 or omitted, it returns the number of duplicates. If num% is –2, it returns the duplicate index of chan% (0 if not a duplicate, 1 for duplicate a and so on).

See also:ChanDuplicate()

EditClear()

In a result view, this command zeros the histogram data and removes any raster information. In a text view, this command 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:EditCut()

EditCopy()

This command copies data from the current view to the clipboard. Time views copy as a bitmap, as a scaleable image and as text in the format set by ExportRectFormat(), ExportTextFormat(), ExportChanFormat() and ExportChanList(). Result and XY views can copy as a bitmap, text and as a scaleable image. Multimedia views copy as a bitmap. Text windows copy as text. You can also copy a string to the clipboard.

```
Func EditCopy({as%|text$});
```

as%   Sets how to copy data when several formats are possible. If omitted, all formats are used. It is the sum of: 1=Copy as a bitmap, 2=Copy as a scaleable image (Windows metafile), 4=Copy as text

text$ A string to place on the clipboard as text. The return value is 4.

Returns It returns 0 if nothing was placed on the clipboard, or the sum of the format specifiers for each format used.

See also:EditSelectAll(), ExportTextFormat(), ExportChanFormat(),
        ExportRectFormat(), ExportChanList(), EditCut(), EditPaste()

EditCut()

This command cuts data from the current view to the clipboard and deletes the original.

```
Func EditCut({as%});
```

as%   This optional argument sets how data is cut and copied to the clipboard. Currently only text may be cut. If omitted, all allowed formats are used.

1   Cut and copy as a bitmap, has no effect
2   Cut and copy as a scaleable image (metafile), has no effect
4   Cut text to clipboard

Returns 0 if nothing was placed on the clipboard, or the sum of the format specifiers for each format that was used. The only possible values now are 0 and 4.

See also:`EditCopy()`,`EditClear()`

EditFind()

In a text view, this command searches from the selection point for the next occurrence of the specified text and selects it if found. This is the same as the Find Text dialog.

```
Func EditFind(find${, dir%{, flags%}});
```

`find$`   The text to search for. May include regular expressions if 4 is added to `flags%`.

`dir%`   Optional. 0=search backwards, 1=search forwards (default), 2=wrap around.

`flags%` Optional, taken as 0 if omitted. The sum of: 1=case sensitive, 2=find complete words only, 4=regular expression search (`dir%` may not be 0).

Returns 1 if found, 0 if not found.

See also:`EditReplace()`

EditPaste()

Either the clipboard contents are inserted into the text at the selection, replacing any selected text or they are returned in a variable. You may only paste into a text window when the clipboard contains text.

```
Func EditPaste({text$});
```

`text$`   If present, any text in the clipboard in returned in this string. If absent, clipboard text is pasted to the current text view.

Returns The number of characters pasted into the view unless `text$` is present when the return value is 1 if the clipboard holds text, 0 if it does not.

EditReplace()

In a text view, this command checks if the selection matches a pattern and replaces it if it does, then it searches for the pattern again. This is the same as the Replace Text dialog.

```
Func EditReplace(find${, repl${, dir%{, flags%}}});
```

`find$`   The text to search for. May include regular expressions if 4 is added to `flags%`.

`repl$`   Optional replacement text, taken as an empty string if omitted.

`dir%`   Optional. 0=search backwards, 1=search forwards (default), 2=wrap around.

`flags%` Optional, taken as 0 if omitted. The sum of: 1=case sensitive, 2=find complete words only, 4=regular expression search (`dir%` may not be 0).

Returns The sum of: 1 if found a new match, 2 if replaced the original selection.

See also:`EditFind()`

EditSelectAll()

This function selects all items in the current 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 Spike2 function.

Returns It returns a string that describes the error.

See also:Debug(), Eval(), Print$(), PrintLog()

| **Eval()** | This evaluates the argument and converts the result into text. The text is displayed in the Script window or Evaluate window message area, as appropriate, when the script ends. This overrides any subsequent script runtime error message. |

```
Proc Eval(arg);
```

arg   A real or integer number or a string.

See also:Debug(), Error$(), Print(), PrintLog()

| **EventToWaveform()** | This function creates a waveform channel from an event channel. The waveform channel contents depend on the frequency of the events. You can convert based on instantaneous or smoothed frequency. VirtualChan() also converts events to a waveform. |

Instantaneous frequency Spike2 calculates instantaneous frequency as the reciprocal of event intervals. Between events, the interval from the last event to the current time is compared with the interval between the last two events. If it is less, the reciprocal of the last interval is used. If it is more, the reciprocal of the interval from the last event is used.
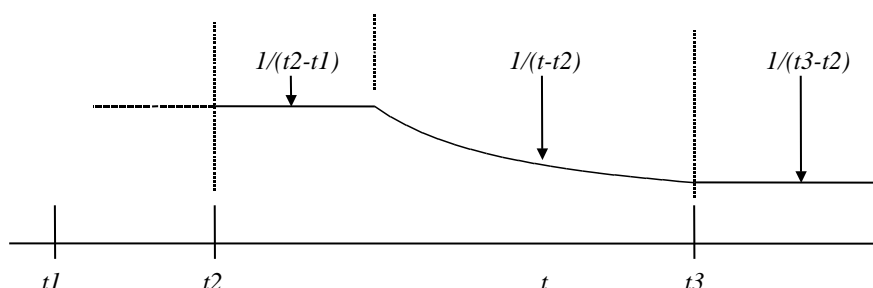
The diagram shows the result for three events at times *t1*, *t2* and *t3*. From *t1* to *t2*, the result depends on events before *t1*. From *t2* to *t2*+(*t2-t1*), the result is set by the previous interval. From this point onwards to *t3*, the output reduces until at *t3* it becomes 1/(*t3-t2*).

Smoothed frequency Spike2 calculates smoothed frequency by replacing (convoluting) each event with a waveform of unit area. The built-in waveforms are symmetrical about the event time. If the event is at time *t*, the waveforms extend from time *t-smooth* to *t+smooth*. You can also provide your own waveforms, and these need not be symmetric.
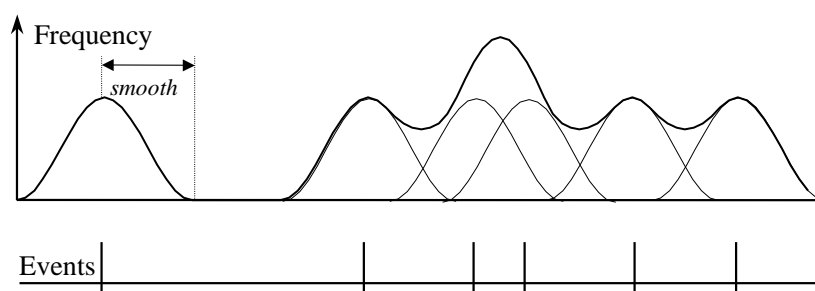
The diagram shows the result of using the built-in raised cosine waveform. The area under the curve for each spike is 1. You would normally use a smoothing period that covered several events to obtain a smooth output.

The command
```
Func EventToWaveform(smooth, eChan%, wChan%, wInt, sTime, eTime,
                     maxF, meanF {,query% {,wave[]|wave%{, nPre}}});
```

smooth   The smoothing period in seconds. Set this 0 or negative for instantaneous frequency, otherwise an event at time t is spread over a time range t-smooth to t+smooth seconds for symmetric functions. If you omit the wave argument, the smoothing function is a raised cosine bell.

eChan%   The channel number to use as a source of event times. If the channel is a marker, and a marker filter is set, only filtered events are visible.

wChan%   The channel number to create as a waveform channel in the range 1 to the maximum allowed channels in the file.

wInt   The desired sampling interval for the new channel in seconds. Values outside the range 0.000001 to 1000 are errors. This interval is rounded to the nearest multiple of the microseconds per time unit set for the file. Use BinSize() to return the actual sampling interval. Before version 4.03, wInt was rounded to the nearest multiple of the base ADC convert interval for the file. Older Spike2 versions will not open the file if you set an interval that they cannot achieve.

sTime   The start time for output of the waveform data.

eTime   The last time for waveform output will be less than or equal to this time. The command processes events from sTime-smooth to eTime+smooth.

maxF   The output is stored as if it were a sampled signal. That is as a 16-bit integer in the range -32768 to 32767. maxF sets the frequency that corresponds to the 32767 value. It is the maximum frequency that can be represented in the result. Frequencies higher than this are limited to maxF.

meanF   This sets the frequency that corresponds to 0 in the 16-bit waveform data. Most users set this to 0. The lowest frequency that can appear in the result is maxF-2*(maxF-meanF). Frequencies lower than this are limited to this value.

         If the result lies in the range freq±delta, you get the best resolution by setting maxF to freq+delta and meanF to freq.

query%   Set this to 0 or omit it to query overwriting an existing channel. If this is present and non-zero, an existing channel is overwritten with no comment.

wave%   This optional argument is used when smooth is greater than 0 and sets the smoothing function: 0=rectangular, 1=triangular, 2=raised cosine, 3=Gaussian (to 4 sigmas). If omitted, raised cosine smoothing is used. These functions are the same as are used for virtual channels.

wave[]   If you provide an array, it is scaled to span smooth seconds. The sum of the waveform values must be in the range 1.2e-38 to 3.4e+38. If nPre is omitted or negative, the smoothing function is symmetrical and you supply the right-hand side only. You can set any array size; the built-in functions use 1000 points. This example matches the raised cosine generated with wave% set to 2.

```
var wave[1000], i%, nPre := -1.0;
const pi := 3.14159;
for i% := 0 to 999 do wave[i%]:=1 + Cos(i%*pi/1000) next;
```

nPre   This sets the index in wave[] that matches the event time for asymmetric smoothing functions. The entire wave[] array is smooth seconds wide. If nPre is omitted or negative, a symmetric function is assumed. This example generates a typical asymmetric smoothing function in the wv[] array:

```
var wv[400],i%, nPre;
const a:=40, b:=80;  'The time constants in points
```

```
for i%:=0 to 399 do wv[i%]:=exp(-i%/a)*(1-exp(-i%/b)) next;
nPre := b*Ln(1+a/b); 'Calculate position of maximum
```

Returns   The function returns 0 unless the user decided not to overwrite a channel, in which case the result is a negative error code. You can also get error -5799 if you run out of memory (unlikely) or disk space.

Uses of EventToWaveform   If you have an event channel of heart beats at 70 beats per minute (1.17 Hz) with a respiration-induced modulation at about 0.25 Hz, you can convert the heartbeats into a waveform with this command. The result would be a constant level of 1.17 Hz, plus a small oscillation of 0.25 Hz. The oscillation can be detected with `SetPower()`.

For a 0.01 Hz resolution with `SetPower()`, you need 100 seconds of data per power spectrum block. With a 1024 point transform size, a `wInt` of 0.1 Hz gives a block size of 102.4 seconds. The `SetPower()` result will have 512 bins, spanning the range 0 to 5.00 Hz in steps of 0.01 Hz. A `smooth` period of 5-10 seconds would be appropriate.

See also: `BinSize()`, `SetPower()`, `VirtualChan()`

Exp()

This function calculates the exponential function for a single value, or replaces a real array by its exponential. 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.

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

ExportChanFormat()

This sets the channel text format used by `ExportTextFormat()`, `FileSaveAs()` and `EditCopy()`. It is equivalent to the Channel type parameters section of the File Dump Configuration dialog. See `ExportTextFormat()` for the command to reset the format.

```
Func ExportChanFormat(type%, synop%, data%{, as%});
```

type%   The channel type to set the format for. Types 2, 3 and 4 share the same settings. Types 1 and 9 also share the same settings.

1 Waveform	3 Event (Evt+)	5 Marker	7 RealMark	9 RealWave
2 Event (Evt-)	4 Level (Evt+-)	6 WaveMark	8 TextMark	

synop%   Set this non-zero to enable synopsis output for this channel type.

data%   Set this non-zero to enable data output for this channel type.

as%   This sets the format for markers and descendent types using the same codes as the `type%` field. It is ignored and can be omitted for waveform and event channels. If omitted or set to 0 or other than 2- 5, the channels are treated as their own type.

Returns   0 if all was OK or a negative error code.

See also: `FileSaveAs()`, `ExportChanList()`, `ExportTextFormat()`

ExportChanList()

This command sets a time range and a channel list to export for use by `FileSaveAs()` and `EditCopy()`. The current view must be a time view. There are two command variants. The first with one or no arguments clears the channel list. The second, with a channel specification, adds a set of channels and a time range to a list., suitable for use with `ExportTextFormat()`. When used with `ExportRectFormat()` or with an external exporter, only the first set of channels and time range is used. To write different segments of a file to separate output files, clear the list with the first command variant before setting each segment with the seconds variant, then output the data.

```
Proc ExportChanList({flags%});
Proc ExportChanList(sTime, eTime, cSpc {,cSpc...});
```

sTime   The start time of the range of data to save.

eTime   The end time of the range of data to save.

cSpc   A channel specifier for the channels to export.

flags%   This affects data written by `FileSaveAs()` with `type%` of 0 and is the sum of:
         1 Time shift data so that the earliest `sTime` appears at time 0 in the output file.
         2 Write RealWave channels as waveform data for backwards compatibility.

See also:`FileSaveAs()`, `ExportChanFormat()`, `ExportTextFormat()`,
        `ExportRectFormat()`, `EditCopy()`

ExportRectFormat()

This command sets the text export format for use by `FileSaveAs()` and `EditCopy()` in a time view. This is an alternative to using `ExportTextFormat()` and generates equally spaced output in time for all channels. If used with no parameters, this will reset the rectangular text output configuration to the default of 100 Hz output, linear interpolation, double-quotes delimiter and tab for the separator. Otherwise at least 2 arguments must be supplied.

```
Proc ExportRectFormat({freq, inter%{, delim${, sep$}}})
```

freq   This sets the number of output points per channel generated per second.

inter%   This sets the interpolation method:
         0   Use the data point nearest to the time.
         1   Linear interpolation between the points either side.
         2   Cubic spline interpolation (waveform data only).
         3   Cubic spline interpolation if drawn splined, else linear (waveforms only).

delim$ This sets the character used to delimit the start and end of text strings. If omitted, text strings are surrounded by quote marks, for example, `"Volts"`.

sep$ The character to separate multiple data items on a line. If omitted, a Tab is used.

The following example exports channels 1, 2 and 3 at 200 Hz from times 10 seconds to 30 seconds as a text file. The times are shifted so that the first data item is at time 0. Waveform channels are mapped to 200 Hz using cubic spline interpolation.

```
ExportChanList(1);              'Clear export list, sets zero shift
ExportChanList(10,30,1,2,3);    'Choose channels 1, 2 and 3
ExportRectFormat(200, 2);       '200 Hz, Cubic spline
FileSaveAs("rect.txt", 1);      'Export as text file, and...
EditCopy();                     '...also copy to the clipboard
```

See also:`FileSaveAs()`, `ExportChanList()`, `EditCopy()`

ExportTextFormat()

This command sets the text export format for use by `FileSaveAs()` and `EditCopy()` in a time view. It is equivalent to the top section of the File Dump Configuration dialog. The form with no arguments resets the dialog to enable all check boxes, sets the columns to 1, sets the string delimiter to a double quote mark, sets the separator to a tab character, and sets all channel types to be treated as themselves.

```
Proc ExportTextFormat(head%, sum%, cols%{, delim${, sep$}});
Proc ExportTextFormat();
```

head%   If this is non-zero, Spike2 will output the header.

sum%   If this is non-zero, Spike2 outputs the channel summary information.

cols%   The number of columns to use when writing waveforms and event times.

delim$ This sets the character used to delimit the start and end of text strings. If omitted, text strings are surrounded by quote marks, for example, "Volts".

sep$ The character to separate multiple data items on a line. If omitted, a Tab is used.

The following example exports channels 1 and 2 from 90.0 to 100.0 seconds and from 110 to 120 seconds as a file 30 seconds long called `fred.smr`.

```
ExportChanList(1);              'Clear export list, sets zero shift
ExportChanList(90, 100, 1, 2);'set channels 1 and 2
ExportChanList(110,120, 1, 2);'add 10 more seconds after 10s gap
ExportTextFormat();             'reset the file dump dialog
ExportChanFormat(1,0,1);        'turn off synopsis for waveform data
ExportTextFormat(0, 0, 1);     'No header or summary
FileSaveAs("fred.smr", 0);     'export to new data file
```

See also:`FileSaveAs()`, `ExportChanList()`, `ExportChanFormat()`, `EditCopy()`

FileApplyResource()

This applies a resource file to the current time, result or XY view. If a time view is duplicated, all other duplicates are deleted, then the resource file is applied, which may create duplicates. The current view handle is not changed. Handles of duplicate views are not preserved, even if the resource file creates the same number of duplicates.

```
Func FileApplyResource(name$);
```

name$ The resource file to apply. You must include the `.s2r` file extension and required path. Remember that `\` in a string must be entered as `\\` or use `/`. If the name is `""` or contains a `"*"` or a `"?"`, the user is prompted for a file.

Returns The number of modified windows (usually 1 unless the resource file generates duplicates), 0 if the resource file did not contain suitable information, -1 if the user cancelled the file dialog, -2 if the file could not be found or -3 if there is any other problem.

See also:`FileGlobalResource()`, `FileOpen()`, `FileSaveResource()`

FileClose()

This is used to close the current window or external file. You can supply an argument to close all windows associated with the current time or result view or to close all the windows belonging to the application.

```
Func FileClose({all% {,query%});
```

all%   This argument determines the scope of the file closing. Possible values are:
    -1 Close all windows 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 window holds unsaved data:
    -1 Don't save the data or query the user

0   Query the user about each window 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.

If the current view is a time view and you have been sampling data, you must call `SampleStop()` before using this command to set a file name. If you do not, this is equivalent to aborting sampling and your data file will not be saved. Use `FileSaveAs()` just before calling `FileClose()` to set a file name.

Do not use `View(fh%).FileClose()` if `fh%` is the current view; you will get an error when Spike2 tries to restore the current view. Use `View(fh%);FileClose()` instead.

See also:`FileOpen(), FileSave(), FileSaveAs(), FileNew(), SampleStop()`

FileComment$()

This function accesses the file comments in the file associated with the current time view. Files have five comment strings. Each string is up to 79 characters in length.

```
Func FileComment$(n% {,new$));
```

n%   The number of the file comment line in the range 1 to 5

new$ If present, the command replaces the existing comment with `new$`.

Returns The comment at the time of the call or a blank string if `n%` is out of range.

See also:`ChanComment$()`

FileConvert$()

This function converts a data file from a "foreign" format into a Spike2 data file. The range of foreign formats supported depends on the number of import filters in the `Spike6\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.

dest$ If 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, Spike2 appends `".smr"`. If you set any other file extension, Spike2 cannot open the file as a data file. If you omit this argument, the converted file is written to the same name and path as the source file with the file extension changed to `.smr`.

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 set to 0 if the file was converted, otherwise it is returned holding a negative error code.

Returns The full path name of the new file or an empty string if there was no output.

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()

FileDate$()

This returns the date when the data file was sampled as a string. Use FileTimeDate() to get the date as numbers. The current view must be a time view. The first 5 arguments are exactly the same as for the Date$() command. The date was not stored before version 4.03. If there is no date, the result is an empty string.

```
Func FileDate$({dayF%{,monF%{,yearF%{,order%{,sep${,offs%}}}}}});
```

offs%   An optional time offset, in seconds, treated as 0 if omitted or negative.

See also:Date$(), FileTimeDate(), FileTime$()

FileDelete()

This deletes one or more files. File names can be specified in full or relative to the current directory. Windows file names look like 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 or use / to separate the parts of the file name.

```
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()

FileGlobalResource()

This function is equivalent to the Global Resources dialog. Please see the documentation for the dialog for a full explanation. The function has two forms:

```
Func FileGlobalResource(flags%, loc%, name${, path$});   Set values
Func FileGlobalResource(&loc%, &name$, &path$);          Get values
```

flags% This is the sum of the following values:
   1   Enable the use of global resources. If unset, no global resources are used.
   2   Only use global resources if a data file has no associated resource file.
   4   Only use if the data file is within the path set by path$.

loc%   The location of the global resource file. 0=the folder that Spike2 was run from, 1=search the data file folder, then the Spike2 folder, 2=the data file folder only.

name$ The name of the global resource file excluding the path and `.s2r` file extension.

path$ The file path within which to use the global resources. If you omit this argument when setting values, the current path does not change.

Returns   When setting values, the return value is 0 or a negative error code. When reading back values, the return value is the `flags%` argument.

See also:`FileApplyResource()`, `FileOpen()`, `FileSaveResource()`

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%   The objects in the current directory to list. The Parent directory is returned with the full path, the others return the name in the current directory. Values are:

-3	Parent directory	2	Output sequence files (`*.pls`)
-2	Sub-directories	3	Spike2 script files (`*.s2s`)
-1	All files	4	Spike2 result view files (`*.srf`)
0	Spike2 data files (`*.smr`)	6	Spike2 configuration files (`*.s2c`)
1	Text files (`*.txt`)	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 but not including the last . 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()`

FileNew()

This is equivalent to the File menu New command. It creates a new window 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%{, upt, tpa%, maxT{, nChan%{, big%}}}});
```

type%   The type of file to create:

  0   A Spike2 data file based on the sampling configuration, ready for sampling. This may open several windows, including the floating command window and the sequencer control panel. Use `SampleStart()` to begin sampling and `SampleStop()` to stop sampling before calling `FileSaveAs()` to give the new file a name. During sampling, type 0 data files are saved in the folder set by the Edit menu preferences or the `FilePathSet()` command. Use `FileSaveAs()` command after `SampleClose()` to move the data file to its final position on disk.

  1   A text file in a window

  2   An output sequence file in a window

  3   A Spike2 script file

  7   An empty Spike2 data file (not for sampling). `upt%`, `tpa%` and `maxT` must be supplied. You can use `ChanNew()`, `ChanWriteWave()`, `ChanSave()`, `MemSave()` or to add data. Use `FileSaveAs()` to name the new file.

  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 or channels, use it (`ViewLink()` can identify them). If this flag is not set, data files extract enough information from the sampling configuration to set the sampling parameters for the data channels.

  4   Show the spike shape setup dialog if there are WaveMark channels in the sampling configuration for a data file. If the spike shape dialog appears, this function does not return until the user closes the dialog.

upt   Only for `type%` = 7. The microseconds per unit time for the new file. This allows values in the range 0.001 to 32767. If `upt` is non-integral values, Spike2 versions prior to 4.02 will not open the file. This sets the time resolution of the new file and the maximum length of the file, in time. There are a maximum of 2^{31}-1 time units in a file, so a 1 μs resolution limits a file to 30 minutes in length.

tpa%   Only for `type%` = 7. The time units per ADC conversion for the new file. You can set values in the range 1 to 32767 (for compatibility with Spike2 sampling use 2 to 32767). The available sampling intervals for waveform data in the new file are `n * upt% * tpa%` microseconds where n is an integer.

maxT   Only for `type%` = 7. This sets the initial file size in time, which sets the value returned by `MaxTime()` until data written to the file exceeds this time.

nChan%   Only for `type%` = 7. This sets the number of channels in the file in the range 32 to 400. If you omit `nChan%`, 32 channels are used. Spike2 version 5 can read up to 400 channels, 4 can read up to 100. Version 3 can only read files with 32.

big%   Only for `type%` = 7. If present and 1 create a big file. Set 0 or omit for normal.

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: `ChanNew()`, `ChanSave()`, `ChanWriteWave()`, `FileOpen()`, `FileSave()`, `FileSaveAs()`, `FileClose()`, `FilePathSet()`, `MaxTime()`, `MemSave()`, `SampleStart()`, `SampleStop()`, `XYAddDAta()`, `XYSetChan()`

FileOpen()

This is the equivalent of the File Open menu command. It opens an existing Spike2 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 contains * or ? (Windows only), the file dialog opens for the user to select a file. For file types 8 and 9 only, name$ can be of the form:

```
"Type 1 (*.fl1)|*.fl1||Type 2 (*.fl2;*.fl3)|*.fl2;*.fl3||"
```

This produces two file types, one with two extensions. There is one vertical bar between the description and the template and two after every template.

type%   The type of the file to open. The types currently defined are:

    0   Open a Spike2 data file in a window
    1   Open a text file in a window
    2   Open an output sequence file in a window
    3   Open a Spike2 script file in a window
    4   Open a result view file in a window
    6   Load configuration file or read configuration from data file
    8   An external text file without a window for use by `Read()` or `Print()`
    9   An external binary file without a window for use by `BRead()`, `BWrite()`, `BSeek()` and other binary routines.
   12   Open an XY view file in a window.

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 4, 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 Spike2. 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.

Returns   For configuration files (type% of 6), the return value is 0 if no error occurs. For other types it is the file view handle (if multiple views open, the handle of the first time view). 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()`, `ViewList()`

FilePath$()

This function gets the current directory. This is the place on disk where file open and file save dialogs start. It can also get the path for created data files and auto-saving.

```
Func FilePath$({opt%});
```

opt%   If 0 or omitted, return the current directory. 1 returns the path for Spike2 data files as set by Edit menu Preferences, 2 gets the path to the Spike2 application, 3 gets the path for automatic file saving as set by the Sampling Configuration Automation tab. From version 7.04, -1 gets the desktop folder, -2 gets the users document folder, -3 gets the Spike2 application folder, -4 gets the user's `Spike7` folder, -5 gets the `Spike7Shared` folder for all users.

Returns A string holding the path or an empty string if an error is detected.

See also: `FileList()`, `FileName$()`, `FilePathSet()`

FilePathSet()

This function sets the current directory/folder, and where Spike2 data files created by `FileNew()` are stored until they are sent to their final resting place by `FileSaveAs()`. There are two command variants:

```
Func FilePathSet(path${, opt%{, make%}});                    'Set path
Func FilePathSet(path${, opt%{, prmpt${, make%}}});    'Open path dialog
```

path$ A string holding the new directory path. The path must conform to the rules for path names on the host system and be less than 255 characters long. If the path is empty or a prompt is set, a dialog opens for the user to select an existing directory/folder or to create a new one.

opt%   If omitted or zero, this sets the current directory. 1 sets the path for Spike2 data files created by `FileNew()`, 2 is ignored, 3 sets the automatic file naming path.

make%   If omitted or 0, only existing directories can be set. If 1, the first command variant can create a directory/folder if all elements of `path$` exist except the last and the second variant allows the user to create a new directory/folder.

prmpt$ Optional prompt. If present a dialog opens for the user to select a directory.

Returns Zero if the path was set, or a negative error code.

See also: `FileList()`, `FileName$()`, `FilePath$()`

FilePrint()

This is equivalent to the File menu Print command. It prints some or all of the current view to the printer that is set for Spike2. If no printer is set, the current system printer is used. In a time or result 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 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 time view, in bins in a result 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, 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 print layout is based on the current view screen format (so could be many pages). The `FilePrintXXXX()` routines can all set the output orientation. If you do not set it, the last print orientation you used in the current Spike2 session is set. If you have not printed, the orientation depends on the current orientation set for the system printer.

See also: `FilePrintScreen()`, `FilePrintVisible()`

FilePrintScreen()

This function is equivalent to the File menu Print Screen command. It prints all visible time, result, XY and text-based views to the current printer on one page. The page positions are proportional to the view positions in the Spike2 application window.

```
Func FilePrintScreen({head${, vTtl%{, box%{, scTxt%{, flags%,
                                                {foot$}}}}}});
```

head$ The page header. If omitted or an empty string, there is no page header.

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.

foot$ The page footer. If omitted or an empty string, there is no page footer.

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 window, this prints the lines in the current selection. If there is no selection, it prints the line containing the cursor.

```
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. You are asked if you wish to save any unsaved data before the application closes. If the user cancels the operation (because there were files that needed saving), the script terminates, but the Spike2 application is left running. Use FileClose(-1, -1) before FileQuit() to guarantee to exit.

```
Proc FileQuit();
```

See also: FileClose()

FileSave()

This function is equivalent to the File menu Save command and saves the current view as a file on disk. If the view has not been saved previously the File menu Save As dialog opens to request a file name. It cannot be used with external text or binary files. It cannot be used with a time view unless it has been sampled and has not been saved.

```
Func FileSave();
```

Returns The function returns 0 if the operation was a success, or a negative error code.

Use with a time view You cannot use this command for a time view unless it has just been sampled; use FileSaveAs() instead. When used with a time view that has just been sampled, if an automatic file name is set, there is no prompt for a file name and the next automatic name is used. However, if automatic filing is enabled, the file is saved as soon as sampling stops, so this function is not needed.

See also: FileSaveAs(), FileSaveResource(), SampleAutoFile()

FileSaveAs()

This function saves the current view in its native format, as text, as an external format or as a picture. It is equivalent to the File menu Save As command. This cannot be used for external text or binary files. This command can also name data files created with `FileNew(7,...)` and after sampling, but read the remarks about time views below.

```
Func FileSaveAs(name${,type%{,yes%{,text${,nCh%|flag%{,exp$|big%}}}}});
```

name$ The output file name including the file extension. If the string is empty or it holds wild card characters * or ? or text$ is not empty, then the File menu Save As dialog opens and name$ sets the initial list of files.

type%   The type to save the file as (if omitted, type -1 is used):
-        -2   Valid for XY and result views only. Opens a File Save As dialog and allows the user to choose the file type to save as.
-        -1   Save in the native format for the view. You can also use this to save and name a Spike2 data file immediately after it has been sampled. This also saves any multimedia data associated with the file. The correct sequence is:
             1. Use `FileNew(0, mode%)` to create a data file
             2. Use `SampleStart()` to start data capture
             3. When capture is over use `SampleStop()` to stop capture, or wait for the `SampleStatus()` to indicate that sampling has finished.
             4. Use `FileSaveAs(name$,-1)` to set the file name
             5. Use `FileClose()` to close the view
             This command can also name a file opened with `FileNew(7,...)`. Once a data file has been named, you must use type 0 to save a selection of channels to a new file; you cannot use type –1 again in a time view.
-        0   Save sections set by `ExportChanList()` to a new Spike2 data file. This will not save associated multimedia files.
-        1   Save the contents of the current view as a text file. If the current view is a time window, Spike2 saves the channels set by `ExportChanList()` in the text format set by `ExportRectFormat()` or `ExportTextFormat()` and `ExportChanFormat()`. See flag% for result and XY views.
-        2   Save a text view as an output sequence file.
-        3   Save a text view as a Script file.
-        4   Save as a result view (result views only).
-        5   Save the screen image of a time, result or XY view as a metafile. The file extension sets the format; use `WMF` for a Windows metafile and `EMF` for an enhanced metafile.
-        6   Save sampling configuration in a configuration file.
-        12   Save as XY view (XY views only).
-        13   Save the time, result, XY or multimedia (video only) view as a bitmap. Video frames are saved at the recorded resolution.
-        14   As 13, but saving in Joint Photographic Expert Group (JPEG) format.
-        15   As 13, but saving in Portable Network Graphics (PNG) format.
-        16   As 13, but saving in Tagged Image File (TIFF) format.
-        100 Types 100 upwards identify external exporters installed in the `export` folder (where you will find additional documentation). Time views export the channels and time range set by `ExportChanList()`. Result and XY views export the data set by flag%.

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.

text$ An optional prompt displayed as part of the file dialog to prompt the user. The File Save As dialog will appear if text$ is not empty.

nCh%   Used when type% is 0 to set the number of channels in the exported file in the range 32-400. If omitted, the file has the same number of channels as the source.

flag%   Used when saving result and XY views as text or to an external exporter to override the normal behaviour, which is to save all channels and all data points. This optional argument has a default value of 0 and is the sum of:

1   Output only visible channels
2   Output only selected channels, or visible channels if no channel is selected
4   Output only data that is in the visible range (use 5 for visible data)

exp$ External exporter settings as a string (see the `export` folder for argument names) of the form: `"name1=value|name2=value|name3=value"`.

big%   Used when `type%` is 0 only. Set 1 for big file, 0 or omit for a normal file.

Returns   The function returns 0 if the operation was a success, or a negative error code.

Use with a newly sampled time view   This command can save a newly sampled time view when `type%` is -1. However, if automatic filing is set with `SampleAutoFile()` or by the sampling configuration, files are saved automatically when sampling ends; using a `type%` of -1 will generate an error.

When automatic file naming is enabled with `SampleAutoName$()` or with the sampling configuration, you can use this command to override the next sequential name, as long as the file has not already been saved and has finished sampling.

See also: `ExportChanFormat()`, `FileClose()`, `FileNew()`, `SampleAutoFile()`

FileSaveResource()

This saves a resource file for the current time, result or XY view. If a time view is duplicated, resources for all the duplicates are saved.

```
Func FileSaveResource({name$|glob%})
```

name$ The resource file to save to. If the name is `""` or contains a `"*"` or a `"?"`, the user is prompted for a file. Any extension in the file name is ignored and the extension is set to `.s2r`. If you omit `name$` and `glob%`, `glob%` defaults to 1.

glob%   An alternative to `name$`. 0=prompt for a file name, 1=save resources to the resource file associated with the view (this may be a global resource file), 2=save to the global resource file (if enabled).

Returns   0 if all was OK, -1 if the user cancelled the File Save dialog, -2 if the file could not be saved for any other reason.

See also: `FileGlobalResource()`, `FileApplyResource()`, `FileSave()`

FileSize()

This function returns the size of the data file associated with the current time view. During sampling this allows for data that is buffered, but not yet written. New in 7.00.

```
Func FileSize();
```

Returns   The size of the file, in bytes, as a real number as it can exceed integer range.

FileTime$()

This returns the time at which sampling started as a string. Use `FileTimeDate()` to get this as numbers. The current view must be a time view. The first 4 arguments are the same as for `Time$()`. Time was not stored in the data file before version 4.03. If there is no time stored, the result is an empty string.

```
Func FileTime$({tBase%{, show%{, amPm%{, sep${, offs%}}}}});
```

offs%   An optional time offset, in seconds, treated as 0 if omitted or negative.

See also: `Time$()`, `FileDate$()`, `FileTimeDate()`

| **FileTimeBase()** | This gets and sets the microseconds per time unit value for the data file associated with the current time view. Changes will likely make nonsense of any derived views; Spike2 does not attempt to fix this. A use of this function is to compensate for time drift between data captured by sampling and data captured by non-synchronised external equipment. |

```
Func FileTimeBase({us});
```

us　　　If present, and in the range 0.001 to 32767, sets the new clock tick unit.

Returns　The microseconds per clock tick at the time of the call.

See also:FileNew()

| **FileTimeDate()** | This returns the sampling start time and date of the current time view file as numbers. FileTime$() and FileDate$() get the result as strings. See TimeDate() for the arguments. If there is no start time, the returned values are all zero. |

```
Proc FileTimeDate(&s%, {&m%,{&h%,{&d%,{&mon%,{&y%,{&wDay%}}}}}});
Proc FileTimeDate(td%[]);
```

See also:Date$(), FileTimeDateSet(), MaxTime(), Time$(), TimeDate()

| **FileTimeDateSet()** | This sets the sampling start time and date of the current time view file. The arguments are the same as for TimeDate() except that td% must have at least 6 elements. |

```
Func FileTimeDateSet(s%, m%, h%, d%, mon%, y%);
Func FileTimeDateSet(td%[]);
```

Returns　0 or a negative error code.

See also:Date$(), FileTimeDate(), MaxTime(), Time$(), TimeDate()

| **FiltApply()** | Applies a set of FIR filter coefficients or a filter in the FIR filter bank to a waveform or RealWave channel in the current time view and places the result in a memory buffer or a disk-based channel. 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 ends of a file and around gaps), extra points are made by duplicating the nearest valid point. |

```
Func FiltApply(n%|coef[], dest%, srce%, sTime, eTime {,flags%});
```

n%　　　Index of the filter in the filter bank to apply in the range -1 to 11, or

coef[]　An array holding a set of FIR filter coefficients to apply to the waveform. The size of the passed array sets the number of coefficients.

dest%　The channel to hold the filtered waveform: either an unused disk channel, a memory channel with the same sample interval as srce% or 0 to create a memory channel. The settings of new channels are copied from the old channel.

srce%　The source waveform or RealWave channel.

sTime　Time to start the output of filtered data. There is no output for areas where there is no input data. If the filter has an even number of coefficients, the output is shifted by half a sample relative to the input.

eTime　The end of the time range for filtered data.

flags%　Optional, default value 0. The sum of the following flag values:

　　　1 Set the destination channel scale and offset values to best represent the output as 16-bit integers. For Waveform output channels, this doubles the processing time and existing data in the output channel that is not overwritten is also

rescaled. If you do not re-scale, the scale and offset are unchanged; the output will be clipped if it exceeds the 16-bit waveform channel output range

2 Create a RealWave destination channel in place of a Waveform channel.

Returns The output channel number or a negative error code if the user clicks Cancel on the progress bar that appears during a long filtering operation or if `dest%` is a disk channel that is in use. Delete existing channels with `ChanDelete(dest%)`.

See also:`FiltAtten()`, `FiltCalc()`, `FiltCreate()`, `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 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 -1 to 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()`, `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 -1 to 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 511.

`dBGot`   If present, set to the attenuation in negative dB 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. If you are passing the result to `FiltApply()`, use `coef[:n]`, not `coef[]` to pass n coefficients.

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 channel 4 (sampled at 200 Hz) with this filter, we may want to calculate the coefficients needed in advance:

```
FiltCalc(0, BinSize(4));
```

This calculates coefficients with the specification of filter 0 for a sampling frequency of 200 Hz with a stop band attenuation of at least the filter 0 attenuation.

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 are limits on the transition gap and on the width of a pass or stop band:

- Minimum transition gap and pass/stop band width is 0.0025 of the sample frequency.
- The maximum transition gap is 0.12 of the sampling frequency.

This function always calculates a set of coefficients, but may temporarily alter the filter specification in order to do it for a particular sampling frequency. This happens:

1. If the transition gap and/or pass and stop band widths are outside their limits, then the limits are applied before calculating the filter.

2. If a filter feature exceeds half the sampling frequency (the Nyquist frequency) it is ignored. In our 50 Hz low pass filter example, if we calculate with a sampling frequency of 80 Hz, the Nyquist frequency is 40 Hz. No frequency above 40 Hz can be represented, 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 to a filter 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()`, `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 -1 to 11.

`new$`   If present, sets the new comment.

Returns The previous comment for the filter at the index.

See also: `FiltApply()`, `FiltCreate()`, `FiltInfo()`, `FiltName$()`, `FiltRange()`

FiltCreate()

This function creates a FIR 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 -1 to 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 the 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(), 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 -1 to 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 real number.

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 -1 to 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 -1 to 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 that 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 that 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 that are calculated. It is important, therefore, to make sure this array is exactly the size that you need. The maximum number of coefficients is 511.

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 the filter is 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 511 (was 255 in versions before 4.01). 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 good 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). If you are passing the result to `FiltApply()`, use `coef[:n]`, not `coef[]` to pass n coefficients.

See also:`ArrFilt()`, `FiltApply()`, `FiltCalc()`, `FIRMake()`, `FIRResponse()`

FIRResponse()

This function retrieves the frequency response of a given filter as amplitude or in dB.

```
Proc 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 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, sets 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 a type not covered by the `FIRQuick()` types, then omit `type%` or set it to -1.

See also:`ArrFilt()`, `FiltCalc()`, `FIRMake()`, `FIRQuick()`

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:

$y = \mathbf{a}_0 \exp(-x/\mathbf{a}_1) + \mathbf{a}_2 \exp(-x/\mathbf{a}_3)\dots$ For an even number of coefficients
$y = \mathbf{a}_0 \exp(-x/\mathbf{a}_1) + \mathbf{a}_2 \exp(-x/\mathbf{a}_3)\dots + \mathbf{a}_n$ 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 $y = \mathbf{a}_0 + \mathbf{a}_1 x + \mathbf{a}_2 x^2 + \mathbf{a}_3 x^3 \dots$ 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:

$y = \mathbf{a}_0 \exp(-\tfrac{1}{2}(x-\mathbf{a}_1)^2/\mathbf{a}_2^2) + \mathbf{a}_3 \exp(-\tfrac{1}{2}(x-\mathbf{a}_4)^2/\mathbf{a}_5^2) + \dots$

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:

$y = \mathbf{a}_0 \sin(\mathbf{a}_1 x + \mathbf{a}_2) + \mathbf{a}_3 \sin(\mathbf{a}_4 x + \mathbf{a}_5) + \dots \{+ \mathbf{a}_{3n}\}$

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 $\pi/2$ 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	Not used
-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 fits 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:

$y = \mathbf{a}_0 \exp(-x/\mathbf{a}_1) + \mathbf{a}_2 \exp(-x/\mathbf{a}_3)\dots$ For an even number of coefficients
$y = \mathbf{a}_0 \exp(-x/\mathbf{a}_1) + \mathbf{a}_2 \exp(-x/\mathbf{a}_3)\dots + \mathbf{a}_n$ For an odd number of coefficients

The fitted coefficients are the $\mathbf{a}_i$; odd numbered $\mathbf{a}_i$ are assumed positive. 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 odd coefficients are similar. 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 (σ), then the error value returned when you iterate to find the best fit is the χ^2 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.

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.

It is important that you give reasonable initial values for the coefficients, especially when you fit more than one exponent. You should limit the odd coefficient values (the time constants) so that they cannot be zero and make sure that multiple exponents do not have overlapping ranges. If two exponents get similar values, the fit is degenerate and will 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.

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 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 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 $(y_{x[i]}-y[i])^2/s[i]^2$ 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 1 if the number of iterations are exhausted and the fit has not converged, 0 if the fit has converged, -1 if the fit is not improving but the result may be OK. Other negative numbers indicate failure.

Select coefficients to fit Normally the command fits all the coefficients, but you can use this 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 array of at least `nCoef%` integers. If `fit%[i]` is 0, coefficient `i` is held constant, otherwise it is fitted. 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 This is a template for using these commands to fit all the coefficients:

```
const nData%:=50;         'set number of data elements
var x[nData%], y[nData%]; 'space for our arrays
var coefs[4],err;         'coefficients and squared error
...                       '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 time constant and range
FitExp(2, 2.0, 0.1, 6);   'set second amplitude and limit range
FitExp(3, .08, .03, .15); 'set second time constant 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:`ChanFit()`, `FitGauss()`, `FitLinear()`, `FitNLUser()`, `FitPoly()`, `FitSin()`

FitGauss()

This command fits multiple Gaussians to 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:

$$y = \mathbf{a}_0 \exp(-\tfrac{1}{2}(x-\mathbf{a}_1)^2/\mathbf{a}_2^2) + \mathbf{a}_3 \exp(-\tfrac{1}{2}(x-\mathbf{a}_4)^2/\mathbf{a}_5^2) + \dots$$

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 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 (σ), then the error value returned when you iterate to find the best fit is the χ^2 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 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 $(y_{x[i]}-\text{y}[i])^2/\text{s}[i]^2$ if `s[]` is used or holding the sum of $(y_{x[i]}-\text{y}[i])^2$ 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 1 if the number of iterations are exhausted and the fit has not converged, 0 if the fit has converged, -1 if the fit is not improving but the result may be OK. Other negative numbers indicate failure.

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   Normally the command fits all the coefficients, but you can use this 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 FitGauss(fit%[]);
```

fit%[] An array of at least nCoef% integers. If fit%[i] is 0, coefficient i is held constant, otherwise it is fitted. 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   This is a template for using these commands to fit all the coefficients:

```
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 coefs[4], err;            'coefficients and 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:ChanFit(), FitExp(), FitLinear(), FitNLUser(), FitPoly(),
        FitSin()

FitLine()

This function calculates the least-squares best-fit line to a set of data points from a time view waveform, RealWave or WaveMark channel or a result view. It fits: $y = \mathbf{m} \, x + \mathbf{c}$ through the data points (x_i, y_i) so as to minimise the error given by: $\text{Sum}_i(y_i - \mathbf{m} \, x_i - \mathbf{c})^2$. In this expression, $\mathbf{m}$ is the gradient of the line and $\mathbf{c}$ is the y axis intercept when x is 0.

```
Func FitLine(chan%, start, finish, &grad{, &inter{, &corr}});
```

chan%   A channel holding suitable data in the current time or result view.

start   The start time for processing in a time view, the start bin in a result view.

finish The end time for processing in a time view, the end bin in a result view. Data at the finish time, or in the finish bin, is included in the calculation.

grad   This is returned holding the gradient of the best fit line (**m**).

inter   Optional, returned holding the intercept of the line with the y axis (**c**).

corr   Optional, 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 time view with a waveform measured in Volts, the units of the gradient are Volts per second and the units of the intercept are Volts. In a result view, the units are y axis units per x axis unit. They are not y units per bin.

See also:ChanFit(), ChanMeasure(), FitLinear(), FitPoly()

FitLinear()	This command fits $y = \mathbf{a}_0 f_0(x) + \mathbf{a}_1 f_1(x) + \mathbf{a}_2 f_2(x)\ldots$ 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 FitLinear(coef[], y[], x[][]{, s{, covar[][]{, r[]{, mR}}}});
```

coef[]   A real array that sets the number of coefficients to fit and 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 functions $f(x)$ at each data point. If there are *nc* coefficients and *nd* data values, this array must be of size at least [*nd*][*nc*]. Viewed as a rectangular grid with the coefficients running from left to right and the data running from top to bottom, the values you must fill in are:

$f_0(x_0)$ $\quad$ $f_1(x_0)$ $\quad$ $f_2(x_0)$ $\quad$ $f_3(x_0)$ $\quad$ … $\quad$ $f_{nc-1}(x_0)$
$f_0(x_1)$ $\quad$ $f_1(x_1)$ $\quad$ $f_2(x_1)$ $\quad$ $f_3(x_1)$ $\quad$ … $\quad$ $f_{nc-1}(x_1)$
$f_0(x_2)$ $\quad$ $f_1(x_2)$ $\quad$ $f_2(x_2)$ $\quad$ $f_3(x_2)$ $\quad$ … $\quad$ $f_{nc-1}(x_2)$
… $\qquad$ … $\qquad$ … $\qquad$ … $\qquad$ … $\qquad$ …
$f_0(x_{nd-1})$ $\quad$ $f_1(x_{nd-1})$ $\quad$ $f_2(x_{nd-1})$ $\quad$ $f_3(x_{nd-1})$ $\quad$ … $\quad$ $f_{nc-1}(x_{nd-1})$

s   This is either a real array holding the standard deviations of the y[] data points, or a real value holding the standard deviation of all data points. If s is omitted or 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 $f(x)$ 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.

        If your fitting functions are not independent of each other, several coefficients may have low r values. The solution is to remove one of the functions from the fit, or to set the mR argument to exclude one of the functions, then fit again. If the remaining coefficients become relevant, you have excluded a function that was a linear combination of the others. If the remaining coefficients 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   This demonstrates how to fit data 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 a trial data set plus noise. We set s to 1.0, so FitLinear() returns the sum of squares of the errors between the fitted and input data. 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 NCOEF%:=2,NDATA%:=50;' coefficient and data sizes
var x[NDATA%][NCOEF%];      ' array of function information
const noise := 0.01;        ' controls how much noise we add
```

```
var data[NDATA%], err := 0;' space for our function and errors
' 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[][]); ' Generate the data, then...
for i%:=0 to NDATA%-1 do    ' ...add noise to it
  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[nd%][0] := v;         ' save the value;
    y[nd%] := a[0] * v;     ' start to build the result
    v := Cos(nd%/20.0);     ' second function
    x[nd%][1] := v;         ' save it
    y[nd%] += a[1]*v;       ' full result
next;
end;
```

See also:ChanFit(), FitExp(), FitGauss(), FitNLUser(), FitPoly(), FitSin()

FitNLUser()

This command uses a non-linear fitting algorithm to fit a user-defined function to a set of data points. The function to be fitted is of the form $y = f(x, \mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2\ldots)$ where the $\mathbf{a}_i$ are coefficients to determine. 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. It 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 to evaluate the error and differentials. If there are n data points, ind% runs 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% that 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 (σ) of the y value of the data points, then the error value returned is the χ^2 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 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 limitations of the script language, you cannot step through the user-defined function; break points in it are ignored when called from FitNLUser(). Using Debug(), Interact(), Toolbar(), DlgShow() or Yield() will cause a runtime error. To check the returned values from the user-defined function, call 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 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 $(y_{x[i]}-y[i])^2/s[i]^2$ 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 1 if the number of iterations are exhausted and the fit has not converged, 0 if the fit has converged, -1 if the fit is not improving but the result may be OK. Other negative numbers indicate failure.

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 fits all the coefficients, but you can use this 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 array of at least nCoef% integers. If fit%[i] is 0, coefficient i is held constant, otherwise it is fitted. 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, NCOEF% := 2;
var x[NDATA%],y[NDATA%],i%;
for i% := 0 to NDATA%-1 do  ' Generate data
   x[i%] := i%;                 ' a:=1, b:=0.05 and add some noise
   y[i%] := exp(-0.05*i%)+(rand()-0.5)*0.01;
   next;

FitNLUser(UserFnc, NCOEF%, NDATA%); 'Link user function
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
```

See also: FitExp(), FitGauss(), FitLinear(), FitPoly(), FitSin()

<div style="border:1px solid black">

FitPoly()

</div>

This command fits $y = \mathbf{a}_0 + \mathbf{a}_1 x + \mathbf{a}_2 x^2 + \mathbf{a}_3 x^3 \dots$ 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 that sets the number of coefficients and returns the coefficient values. 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 chi-squared if s[] or s is given (and non-zero), otherwise it returns the sum of squares of the errors between the raw and fitted data.

An example This example generates test data, adds random noise, then fits a polynomial to the data.

```
const NCOEF% := 5;         ' number of coefficients
const NDATA%:=50;          ' number of data points
var y[NDATA%], x[NDATA%];  ' space for x and y values for fit
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;
```

See also:ChanFit(), FitExp(), FitGauss(), FitLinear(), FitSin()

| FitSigmoid() | This command fits a single Sigmoid function to 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: |

$$y = \mathbf{a}_0 + (\mathbf{a}_1 - \mathbf{a}_0) / (1 + \exp((\mathbf{a}_2 - \mathbf{x}) / \mathbf{a}_3))$$

The fitted parameters (coefficients) are the $\mathbf{a}_i$. You can only fit 1 Sigmoid. The commands to implement this are:

Set up the problem The first command sets the number of Sigmoids to fit and the data set to be fitted. You must call this function before you call any of the others.

```
func FitSigmoid(nCoef%, y[], x[]{, s[]|s});
```

nCoef% The number of coefficients to fit. The only legal value is 4 for one Sigmoid.

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 a Sigmoid, it is usual that the two levels (a0 and a1) are easy to determine. If you can set the levels and limit them so that they cannot overlap, the fit usually will proceed without any problems. 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 FitSigmoid(coef%, val{, lo, hi});
```

coef%   The coefficient to set. The first coefficient is number 0, the last 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 two levels, there should be no problems fitting a Sigmoid.

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 FitSigmoid(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 (yx[i]-`y[i]`) `^2`/`s[i]` `^2` if `s[]` is used or holding the sum of (yx[i]-`y[i]`) `^2` if `s[]` is not used, where yx[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 1 if the number of iterations are exhausted and the fit has not converged, 0 if the fit has converged, -1 if the fit is not improving but the result may be OK. Other negative numbers indicate failure.

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 Normally the command fits all the coefficients, but you can use this 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 FitSigmoid(fit%[]);
```

fit%[] An array of at least `nCoef%` integers. If `fit%[i]` is 0, coefficient `i` is held constant, otherwise it is fitted. 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 This is a template for using these commands to fit all the coefficients:

```
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 coefs[4], err;                 'coefficients and error squared
...                                'in here goes code to get the data
FitSigmoid(4, y[], x[], s[]);      'fit one gaussian
FitSigmoid(0, 1.0, 0.2, 4);        'set base level and limit range
FitSigmoid(1, 20, 15, 25);         'set end level and range
FitSigmoid(2, 8);                  'initial 50% point in X units
FitSigmoid(3, 0.5);                'initial slope
repeat
   DrawMyData(coefs[], x[], y[]);  'Some function to show progress
until FitSigmoid(coefs[], err, 1) < 1;
DrawMyData(coefs[], x[], y[]);     'Show the final state
```

See also:ChanFit(), `FitExp()`, `FitGauss()`, `FitLinear()`, `FitSin()`

FitSin()

This command fits 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:

$$y = a_0 \sin(a_1 x + a_2) + a_3 \sin(a_4 x + a_5) + \ldots \{+ a_{3n}\}$$

The fitted coefficients are the a_i. Angles are evaluated in radians. You can fit up to 3 sinusoids and an optional offset. Although the function is given in terms of sine functions, you can easily convert to cosines by subtracting $\pi/2$ from the phase angle (a_2, a_5, 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 or 4, 7 and 10 to include an offset as the last coefficient.

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 that 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 (σ), then the error value returned when you iterate to find the best fit is the χ^2 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 complete.

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 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 1 if the number of iterations are exhausted and the fit has not converged, 0 if the fit has converged, -1 if the fit is not improving but the result may be OK. Other negative numbers indicate failure.

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 array of at least nCoef% integers. If fit%[i] is 0, coefficient i is held constant, otherwise it is fitted. 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. 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 command (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 coefs[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
```

See also:FitExp(), FitGauss(), FitLinear(), FitNLUser(), FitSin()

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 functions 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 0 or a negative error code for an array or the next lower integral value.

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 multimedia and spike shape windows.

```
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, &flags%{, style%{,&fore%{,&back%}}});
```

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.

flags% Returned holding the sum of the style values: 1=Italic, 2=Bold, 4=Underline, 8=Force upper case, 16=force lower case. Only one of 8 or 16 will be returned.

style% Text-based views have all have default style (32) that is used as the basis of all other styles, plus a number of additional styles that are used to highlight

keywords and the like in output sequencer and script windows. The style codes for other styles start at 0 and run upwards. You can find a list of styles for each view type in the Edit Preferences General tab in the Editor settings section. If you omit `style%` the settings for the default style (32) are returned.

In a time view, style 0 (or omitting style) gets the view font, style 1 gets the font for Vertical Markers.

`fore%`   This value returns the foreground colour of the style in text views. Bits 0-7 hold the red intensity, bits 8-15 hold the green and bits 16-23 hold the blue.

`back%`   This value returns the background colour of the style in text views.

Returns   The function returns 0 if all was well or a negative error code. If an error occurs, the variables are not changed.

The arguments from `style%` onwards and the `flag%` values 4, 8 and 16 only apply to text-based views and are new in version 6.

See also:`FontSet(), TabSettings()`

FontSet()

This function sets the font for the current view. Text-based views (text, sequencer and script) normally avoid proportionally spaced fonts as they did not display correctly before Spike2 version 6. Arguments `fore%` and `back%` are for text-based views.

```
Func FontSet(name$|code%, size, flags%{,style%{,fore%{,back%}}});
```

`name$`   A string holding the font name to use or you can specify a font code:

`code%`   This is an alternative method of specifying a font. We recognise these 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. Your system may limit the allowed range.

`flags%`   The sum of the style values to set: 1=Italic, 2=Bold, 4=Underline, 8=Force upper case, 16=force lower case. If both 8 and 16 are set, 16 is ignored. Values from 4 upwards are only supported by text-based views.

`style%`   Text-based views have a default style (32) plus a number of additional styles that are used to highlight keywords in output sequencer and script windows. The style codes for other styles start at 0 and run upwards. You can find a list of styles for each view type in the Edit Setup dialog. If you omit `style%`, the default style is changed. Set the value -1 to set all styles to the values you give.

In other views, style 0 (or omitting style) sets the view font. In a time view, style 1 sets the font for Vertical Markers.

`fore%`   This sets the style foreground colour or is set to -1 to make no change. Bits 0-7 hold the red intensity, bits 8-15 hold the green and bits 16-23 hold the blue. Bits 24-31 are 0. It is convenient to code this as a hexadecimal number, for example:
```
const red%:=0x0000ff, green%:=0x00ff00, blue%:=0xff0000;
const gray%:=0x808080, yellow% :=0x00ffff;
```

`back%`   This sets the background colour of the style in the same format as `fore%`.

Returns   The function returns 0 if the font change succeeded, or a negative error code.

See also:`FontGet(), TabSettings()`

Frac()

Returns the fractional part of a real number or converts a real array to its fractional parts.

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

x   A real number or an array of real numbers.

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). Frac(4.7) is 0.7, Frac(-4.7) is -0.7.

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

FrontView()

The front view is the time, result, XY or text-based view that decides which menu appears in the main window. You can use this command to get the front view, or to set it. When a view becomes the front view, it is moved to the front and made the current view. If an invisible or iconised view is made the front view, it is made visible automatically, (equivalent to WindowVisible(1)).

You can also use this command to move other windows (for example the multimedia and Spike shape windows) to the top and make them the current view, but such windows are not returned by this command. Use FocusHandle() to return these windows.

```
Func FrontView( {vh%} );
```

vh%   Either 0 or omitted to return the front view handle, a view handle to be set, or -n, meaning the nth duplicate of the time 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: FocusHandle(), View(), WindowVisible()

GammaP(), GammaQ()

These functions return the incomplete gamma function $P(a, x)$. It is defined as:

$$P(a,x) = \frac{\int_0^x e^{-t}t^{a-1}dt}{\Gamma(a)} \text{ where } \Gamma(a) = \int_0^\infty t^{a-1}e^{-t}dt$$

This is named GammaP() in the script. We also define the complement of this function, GammaQ(a,x), which is 1.0-GammaP(a,x). From them are obtained the error function, the cumulative Poisson probability function and the Chi-squared probability function.

The error function $erf(x) = \frac{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).

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);
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   These functions return the incomplete Gamma function and the complement of the incomplete Gamma function.

See also:LnGamma()

Grid()

This function turns the background grid on and off for the current time, result or XY view and returns the state of the grid. Separate control of x and y was added at version 7.02.

```
Func Grid({on%});
```

on%   Optional, sets the grid state. Omit or set -1 for no change, 0 = no grid, 1 = both x and y grid, 2=x grid only, 3=y grid only.

Returns   The state of the grid at the time of the call as 0-3 or a negative error code. Changes invalidate the data area and it will redraw at the next opportunity.

See also:XAxis(), XScroller(), YAxis()

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.

See also:ViewLineNumbers()

HCursor()

This function returns the 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. It is also used with spike shape window to set trigger levels and can read back the last cursor levels associated with Edit WaveMark views.

```
Func HCursor(num% {,where {,chan%}});
```

num%   The cursor to use. It is an error to attempt this operation on an unknown cursor.

        In a spike shape window, cursor 1 is the low trigger level, 2 is the high trigger, 3 is the low limit (if enabled), 4 is the high limit (if enabled). To set the cursor for trace n (0-3) add 4*n to the cursor number.

where   If this parameter is given it sets the new position of the cursor.

chan%   If this parameter is given, it sets the channel number. In XY or spike shape views, you should omit this argument as it is ignored.

Returns   The function returns the position of the cursor at the time of the call.

See also:HCursorChan(), HCursorDelete(), HCursorLabel(),
        HCursorLabelPos(), HCursorNew(), HCursorRenumber()

HCursorChan()

This function returns the channel number that a particular cursor is 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. XY views return 1 as the channel number.

See also:HCursor(), HCursorDelete(), HCursorNew(), HCursorRenumber()

HCursorDelete()

This deletes the designated horizontal cursor. Deleting an unknown cursor has no effect.

```
Func HCursorDelete({num%});
```

num%   The number of the cursor to delete. If this is omitted, the highest numbered cursor is deleted. Set -1 to delete all horizontal cursors.

Returns   The number of the deleted cursor or 0 if no cursor was deleted.

See also:HCursor(), HCursorChan(), HCursorLabel(), HCursorLabelPos(), HCursorNew(), HCursorRenumber()

HCursorExists()

Use this function to determine if a horizontal cursor exists.

```
Func HCursorExists(num%);
```

num%   The cursor number in the range 1-4.

Returns   0 if the cursor does not exist, 1 if it does.

See also:HCursorNew(), CursorExists()

HCursorLabel()

This gets and optionally sets the horizontal cursor label style for the view or for a cursor. You can label cursors with a position and/or the cursor number, or with user-defined text. There are two variants: the first sets styles and labels; the second reads back user-defined labels.

```
Func HCursorLabel({style%{, num%{, form$}}});
Func HCursorLabel(&form$, num%);
```

style%   Set 0=None, 1=Position, 2=Number, 3=Both, 4=User-defined. Omit for no change. Style 4 is used with a format string. Styles 0-3 set the styles of cursors selected by num% and the view style for new cursors if num% is -1 or omitted. Style 4 is applied to cursors set by num%; it does not set the view style.

num%   1-9 to select a single cursor or to return the label string. Omit num% or set it less than 1 (use -1 as we may add cursor 0 in the future) for all cursors and to get and set the style for new cursors.

form$ The user-defined 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.

Returns   A cursor style before any change as 0-4 if a single cursor is selected, or the view cursor style as 0-3. If style% is omitted or not 0-3, the current view cursor style is not changed.

See also:HCursorLabelPos(), HCursorNew(), HCursorRenumber()

HCursorLabelPos()

This lets you set and read the position of the cursor label.

```
Func HCursorLabelPos(num% {,pos});
```

num%　The cursor number. Setting a silly number 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.

See also:HCursor(), HCursorChan(), HCursorDelete(), HCursorLabel(),
　　　　HCursorNew(), HCursorRenumber()

HCursorNew()

This function creates a new horizontal cursor in a time or result view and assigns it to a channel. You can create up to 9 horizontal cursors.

```
Func HCursorNew(chan% {,where});
```

chan%　A channel for the new cursor. If the channel is hidden, the cursor is not visible. You should use a channel number of 1 for XY views.

where　An optional argument setting the cursor position. If omitted, the cursor is set to a y axis position proportional to the cursor number 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:HCursor(), HCursorChan(), HCursorDelete(), HCursorLabel(),
　　　　HCursorLabelPos(), HCursorRenumber()

HCursorRenumber()

This command renumbers the cursors from bottom to top. There are no arguments.

```
Func HCursorRenumber();
```

Returns　The number of cursors found in the view.

See also:HCursor(), HCursorChan(), HCursorDelete(), HCursorLabel(),
　　　　HCursorLabelPos(), HCursorNew()

Help()

This command lets you display help information from the standard Spike2 help file or from a help file of your choosing. You can also use Help(0) to close any open help file.

```
Func Help(topic%|topic$ {,file$});
```

topic%　A numeric code for the help topic. These codes are assigned by the help system author. If topic% is zero, this closes any open help file and changes the default help file to file$ or sets it back to the standard help file if file$ is omitted or empty.

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 Spike2 help file is used. If this holds a filename, this filename is used as the help file. If topic% is present and zero, this sets the name of the help file to use for future help requests until the script terminates.

Returns　1 if the help topic was found, 0 if it was not found, -1 if no help file was found. Help(0, file$) returns 0 if the new file exists and -1 if it does not.

The Windows SDK has some help-authoring tools, and third-party tools are available if you wish to generate your own .chm files.

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 were originally 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:

$$y[\text{n}] = \sum_{i=0}^{N} a_i\, x[\text{n-}i]/G + \sum_{i=1}^{M} b_i\, y[\text{n-}i]$$

where the $x[\text{n}]$ are the sequence of input data values, the $y[\text{n}]$ 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, twice the order for bandpass and bandstop designs, and is 2 for resonators and notch filters. 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:

- FIR filters are unconditionally stable. IIR filters are prone to stability problems if very narrow features (<0.0001 of the sample rate) are used. Problems increase at high filter orders. Filters report if they are likely to be unstable.
- 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.
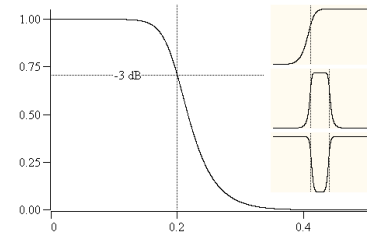
Bessel   An analogue Bessel filter has the property that the group delay is maximally flat, which means that it tends to preserve the shape of a signal passed through it. This leads to filters with a gentle cut-off. When digitised, the constant group delay property is compromised; the higher the filter order, the worse the group delay. The example shows a fifth order low pass filter at 0.2 of the sample rate.
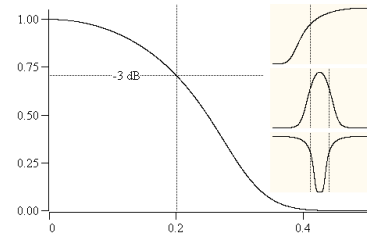
Chebyshev type 1   Filters of this type are based on Chebyshev polynomials and have the fastest transition between the pass band and the stop band for a given ripple in the pass band and no ripples in the stop band. In the example, the ripple has been set to 3 dB, to match the other examples, though you would normally choose less ripple than this.
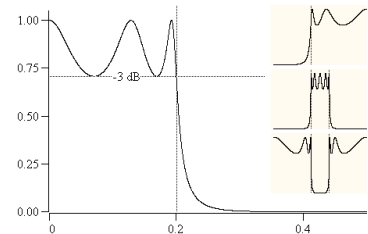
Chebyshev type 2   Filters of this type are defined by the start of the stop band and the stop band ripple. The filter has the fastest transition between the pass and stop bands given the stop band ripple and no ripple in the pass band. The example shows a fifth order filter with a 40 dB stop band ripple and with the stop band starting at 0.2 of the sample rate.
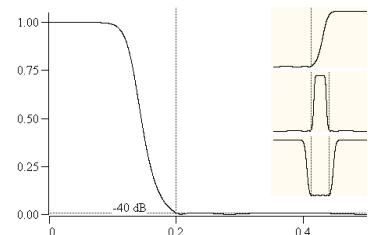
Notch   Notch filters are defined by a centre frequency and a q factor. q is the width of the stop band at the −3 dB point divided by the centre frequency: the higher the q, the narrower the notch. Notch filters are often used to remove mains hum, but if you do this, you will likely need to set notches 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.
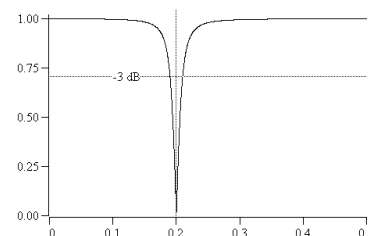
Resonator   A resonator is the inverse of a notch. It is defined in terms of a centre frequency and a q factor. q is the width of the pass band at the −3 dB point divided by the centre frequency: the higher the q, the narrower the resonance. Resonators are sometimes used as alternatives to a narrow bandpass 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.
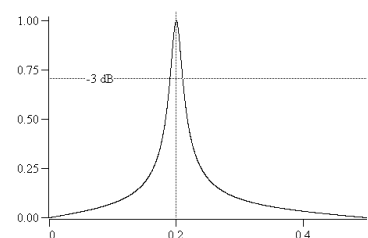
Common command	The `IIRBp()`, `IIRBs()`, `IIRLp()`, `IIRHp()`, `IIRReson()` and `IIRNotch()`
variants	commands are all independent and they each remember the last filter you created and the
	filter state after the last filtering operation. They all support the following variants:

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 set to 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 set to 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, `arr[r][2]`. The frequency response is returned as complex numbers in the columns; `arr[][0]` holds the real part and `arr[][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.

4   The same as 3 except that the results are returned as the amplitude response in `arr[][0]` and the phase response in `arr[][1]`. If the real and imaginary parts of the response are r and i, `arr[][0]` holds `sqrt(r*r+i*i)` and `arr[][1]` holds `atan(i, r)`. The return value is the maximum returned amplitude response.

5   Returns the number of filter coefficients ($N+1$) 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. N is the filter order for low pass and high pass, twice the order for band pass and band stop, and 2 for resonators and notch filters.

6   Returns the number of filter coefficients ($N+1$) to apply to the filter output values and fills in `arr[]` with these values. These correspond to the b_i in the filter expression. 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 of *N* (see get% = 5) rows and 2 columns. The return value is the number of poles in the s-plane. `arr` is filled in with the complex poles with the real part in `arr[][0]` and the imaginary part in `arr[][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.

12 Returns the stability, being the distance of nearest pole to the unit circle. Stability values greater than 1e-12 generally produce acceptable results.

Apply existing filter   This command applies the current filter to an array of equally spaced data

```
Func IIRxxxx(data[]{, flags%{, save[]}});
```

data   An array of data to filter.

flags% Optional, taken as 0 if omitted. Add:

1   Filter backwards. IIR filters introduce phase shifts; running a filter forwards then backwards cancels the phase shift at the expense of a non-causal filter.

2   Treat `data[]` as a continuation of the last filtering operation (only do this if it really is a continuation, otherwise the results are nonsense). If you apply the

filter backwards, you must present the data blocks backwards. To interleave multiple data streams you must use the optional `save[]` argument.

save[]   Optional. This real array preserves the filter state so that you can interleave filtering with the same filter between multiple data streams. The minimum size of the array depends on the filter type and order. It is 2*order+2 for `IIRLp()` and `IIRHp()`, 4*order+2 for `IIRBp()` and `IIRBs()` and 6 for `IIRReson()` and `IIRNotch()`. State is loaded from `save[]` before filtering when 2 is added to `flags%`; it is always updated after filtering. Filling `save` with zeros resets all memory of filtering.

See also:`IIRBp()`, `IIRBs()`, `IIRHp()`, `IIRLp()`, `IIRNotch()`, `IIRReson()`

IIRApply()

Applies a filter in the IIR filter bank to a waveform or RealWave channel in the current time view and places the result in a destination memory buffer or a disk-based channel. The output can be written as either a waveform or as a RealWave channel.

```
Func IIRApply(index%, dest%, srce%, sTime, eTime {,flags%});
```

index%   Index of the filter in the filter bank to apply in the range -1 to 11

dest%   The channel to hold the filtered waveform: either an unused disk channel, a memory channel with the same sampling frequency as `srce%` or 0 to create a compatible memory channel and place the filtered waveform in it. When a new channel is created, the channel settings are copied from the old channel.

srce%   The source waveform or RealWave channel. There must be at least half the number of sampling coefficients worth of data points before `sTime` if the output is to start at `sTime`. Similarly, the channel must extend for the same number of data points beyond `eTime` if the output is to extend to `eTime`.

sTime   Time to start the output of filtered data. There is no output for areas where there is no input data. If the filter has an even number of coefficients, the output is shifted by half a sample relative to the input.

eTime   The end of the time range for filtered data.

flags%   Optional, default value 0. The sum of the following flag values:

     1   Optimises the destination channel scale and offset values to give the best possible representation of the output as 16-bit integers. For Waveform output channels, this doubles the processing time and existing data in the output channel that is not overwritten is also rescaled. If you do not re-scale, the channel's scale and offset are unchanged. However, you run the risk of the output being clipped to the 16-bit range allowed for a waveform channel

     2   Create a RealWave destination channel in place of a Waveform channel.

Returns   The channel number that the output was written to 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 or if `dest%` is a disk channel that is in use. Delete an existing channel with `ChanDelete(dest%)`.

See also:`ChanDelete()`, `IIRComment$()`, `IIRCreate()`, `IIRInfo()`, `IIRName$()`

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[]{, flags%{, save[]}});
Func IIRBp(get%{, arr[]{[]}});
```

data   An array of data to filter. If there are only 1 or 2 arguments, the last created band pass filter is used. Otherwise, the filter defined by the remaining arguments is used forwards. Replace data with 0 to create a filter without applying it.

flags% Optional, taken as 0 if omitted. Add 1 to run the filter backwards, 2 to treat data[] as a continuation of the last filtering operation.

save[] Optional, with a size of at least 4*order%+2. It is used to save the filter state to allow use with the continued data flag with multiple data streams.

lo   The low corner frequency of the band stop filter as a fraction of the sample rate in the range 0.000001 to 0.499998. 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 as a fraction of the sampling rate in the range lo+0.000001 to 0.499999. 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. 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 ripple value must be in the range 0.01 to 1000 dB.

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   An optional 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 A negative return indicates an error. Commands that make or apply filters return 0 for success or 1 if the filter stability is less than 1e-12. Other command return values are included in the description of the get% argument or are 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 IIRBs(data[]|0, lo, hi, order%{, type%{, ripple}});
Func IIRBs(data[]{, flags%{, save[]}});
Func IIRBs(get%{, arr[]{[]}}});
```

data   An array of data to filter. If there are only 1 or 2 arguments, the last created band stop filter is used. Otherwise, the filter defined by the remaining arguments is applied forwards. Replace data with 0 to create a filter without applying it.

flags% Optional, taken as 0 if omitted. Add 1 to run the filter backwards, 2 to treat data[] as a continuation of the last filtering operation.

save[] Optional, with a size of at least 4*order%+2. It is used to save the filter state to allow use with the continued data flag with multiple data streams.

lo   The low corner frequency of the band pass filter as a fraction of the sample rate in the range 0.000001 to 0.499998. For Chebyshev filters, this is the point at which the attenuation reaches the ripple value; for other filters, it sets the –3 dB point.

hi   The high corner frequency of the band pass filter as a fraction of the sampling rate in the range `lo+0.000001` to 0.499999. 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. 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 ripple value must be in the range 0.01 to 1000 dB.

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   An optional 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   A negative return indicates an error. Commands that make or apply filters return 0 for success or 1 if the filter stability is less than 1e-12. Other command form return values are included in the description of the `get%` argument or return 0.

See also:`FIRMake(), IIRBp(), IIRHp(), IIRLp(), IIRNotch(), IIRReson()`

IIRComment$()

This function gets and sets the comment associated with an IIR filter in the filter bank.

```
Func IIRComment$(index% {, new$});
```

index%   Index of the filter in the filter bank to use in the range -1 to 11.

new$ If present, sets the new comment.

Returns   The previous comment for the filter at the index.

See also:`IIRApply(), IIRInfo(), IIRName$()`

IIRCreate()

This creates an IIR filter description and adds it to the filter bank.

```
Func IIRCreate(index%, type%, model%, order%, fr1{,fr2{,extra}});
```

index%   Index of the filter in the filter bank in the range -1 to 11.

type%   Sets the filter type as: 0=Low pass, 1=High pass, 2=Band pass, 3=Band stop.

model%   Sets the filter model: 0=Butterworth, 1=Bessel, 2=Chebyshev type 1, 3=Chebyshev type 2, 4=Resonator.

order%   Sets the filter order in the range 1-10. Resonators always set an order of 2.

fr1   Sets the corner frequency for low pass, high pass filters, the centre frequency for resonators, and the low corner frequency for band pass and band stop filters.

fr2   Sets the upper corner frequency for band pass/stop filters, otherwise ignored.

extra   Sets the ripple for Chebyshev filters in the range 0.01 to 1000 and the Q factor for resonators in the range 1 to 10000.

Returns   0 if OK or a negative error code if the operation failed.

See also:`IIRApply(), IIRComment$(), IIRCreate(), IIRName$()`

| **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[]{, flags%{, save[]}});
Func IIRHp(get%{, arr[]{[]}});
```

data   An array of data to filter. If there are only 1 or 2 arguments, the last created high pass filter is used. Otherwise, the filter defined by the remaining arguments is applied forwards. Replace `data` with 0 to create a filter without applying it.

flags%   Optional, taken as 0 if omitted. Add 1 to run the filter backwards, 2 to treat `data[]` as a continuation of the last filtering operation.

save[]   Optional, with a size of at least 2*order%+2. It is used to save the filter state to allow use with the continued data flag with multiple data streams.

edge   The corner frequency of the high pass filter as a fraction of the sample rate in the range 0.000001 to 0.499999. For Chebyshev filters, this is the point at which the attenuation reaches the `ripple` value; for other filters, it sets the –3 dB point.

order%   The order of the filter in the range 1 to 10.

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 ripple value must be in the range 0.01 to 1000 dB.

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   An optional 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   A negative return indicates an error. Commands that make or apply filters return 0 for success or 1 if the filter stability is less than 1e-12. Other command form return values are included in the description of the `get%` argument or return 0.

See also:FIRMake(), IIRBp(), IIRBs(), IIRLp(), IIRNotch(), IIRReson()

| **IIRInfo()** | Retrieves information about an IIR filter in the bank. |

```
Func IIRInfo(index%, &model%, &order%, &fr1{, &fr2{, &extra}});
```

index%   Index of the filter in the filter bank in the range -1 to 11.

model%   Returned as the filter model: 0=Butterworth, 1=Bessel, 2=Chebyshev type 1, 3=Chebyshev type 2, 4=Resonator.

order%   Returned as the filter order in the range 1-10. Resonators always return 2.

fr1   Returned as the corner frequency for low and high pass filters, as the low corner for band pass and band stop filters and as the centre frequency for resonators.

fr2   Returned as the upper corner frequency for band pass and band stop filters, otherwise set the same as `fr1`.

extra   Returned as the ripple for Chebyshev filters and as the Q factor for resonators.

Returns   The type of the filter as 0=Low pass, 1=High pass, 2=Band pass, 3=Band stop.

See also:IIRApply(), IIRComment$(), IIRCreate(), IIRName$()

| **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[]{, flags%{, save[]}});
Func IIRLp(get%{, arr[]{[]}});
```

data   An array of data to filter. If there are only 1 or 2 arguments, the last created low
           pass filter is used. Otherwise, the filter defined by the remaining arguments is
           applied forwards. Replace data with 0 to create a filter without applying it.

flags%   Optional, taken as 0 if omitted. Add 1 to run the filter backwards, 2 to treat
          data[] as a continuation of the last filtering operation.

save[]   Optional, with a size of at least 2*order%+2. It is used to save the filter state to
          allow use with the continued data flag with multiple data streams.

edge   The corner frequency of the low pass filter as a fraction of the sample rate in the
           range 0.000001 to 0.499999. For Chebyshev filters, this is the point at which the
           attenuation reaches the ripple value; for other filters, it sets the −3 dB point.

order%   The order of the filter in the range 1 to 10.

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 ripple value must be in the range 0.01 to 1000 dB.

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   An optional 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   A negative return indicates an error. Commands that make or apply filters return
          0 for success or 1 if the filter stability is less than 1e-12. Other command form
          return values are included in the description of the get% argument or return 0.

See also:FIRMake(), IIRBp(), IIRBs(), IIRHp(), IIRNotch(), IIRReson()

IIRName$()

This function gets and/or sets the name of an IIR filter in the filter bank.

```
Func IIRName$(index% {, new$});
```

index%   Index of the filter in the filter bank to use in the range -1 to 11.

new$ If present, sets the new name.

Returns   The previous name of the filter at that index.

See also:IIRApply(), IIRComment$(), IIRInfo()

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[]{, flags%{, save[]}});
Func IIRNotch(get%{, arr[]{[]}});
```

data   An array of data to filter. If there are only 1 or 2 arguments, the last created
           notch filter is used. Otherwise, the filter defined by the remaining arguments is
           applied forwards. Replace data with 0 to create a filter without applying it.

flags%   Optional, taken as 0 if omitted. Add 1 to run the filter backwards, 2 to treat
          data[] as a continuation of the last filtering operation.

save[] Optional, with a size of at least 6. It is used to save the filter state to allow use with the continued data flag with multiple data streams.

fr   The frequency of the notch. This is expressed as a fraction of the sample rate and is limited to the range 0.000001 to 0.499999.

q   The q factor for the notch in the range 1 to 10000; the higher the q, the narrower the notch. If Flo and Fhi are the frequencies of the –3 dB points either side of the notch, q is fr/(Fhi-Flo). 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   An optional 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 A negative return indicates an error. Commands that make or apply filters return 0 for success or 1 if the filter stability is less than 1e-12. Other command form return values are 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[]{, flags%{, save[]}});
Func IIRReson(get%{, arr[]{[]}});
```

data   An array of data to filter. If there are only 1 or 2 arguments, the last created resonator filter is used. Otherwise, the filter defined by the remaining arguments is applied forwards. Replace data with 0 to create a filter without applying it.

flags%   Optional, taken as 0 if omitted. Add 1 to run the filter backwards, 2 to treat data[] as a continuation of the last filtering operation.

save[] Optional, with a size of at least 6. It is used to save the filter state to allow use with the continued data flag with multiple data streams.

fr   The centre frequency of the resonator. This is expressed as a fraction of the sample rate and is limited to the range 0.000001 to 0.499999.

q   The q factor for the resonator in the range 1 to 10000; the higher the q, the narrower the resonance. If Flo and Fhi are the frequencies of the –3 dB points either side of the resonance, q is fr/(Fhi-Flo). 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   An optional 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 A negative return indicates an error. Commands that make or apply filters return 0 for success or 1 if the filter stability is less than 1e-12. Other command form return values are included in the description of the get% argument or return 0.

See also:FIRMake(), IIRBp(), IIRBs(), IIRHp(), IIRLp(), IIRNotch()

Inkey()

This is provided for compatibility with the MS-DOS version of Spike2. Do not use it in new scripts. It returns the ASCII code for the key pressed, or -1 if no key was pressed. In some cases Spike2 absorbs keystrokes, for example if you are sampling and the current window is the sampling time window all keystrokes are taken as markers.

```
Func Inkey();
```

Returns The key code or -1 if there is no pending key. The codes are:

```
  1-31   Ctrl+ABC...XYZ[\]^_          65-95   ABC...XYZ[\]^_
    32   space                          96   `
  33-47  !"#$%&'()*+,-./              97-126  abc...xyz{|}~
  48-64  0123456789:;<=>?@              127   Rubout
```

See also: Interact(), Keypress(), Toolbar(), ToolbarSet()

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{, pre%}}});
```

text$ A string holding a prompt for the user. If the string includes a vertical bar, the text before the vertical bar is used as the window title.

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.

pre%   If present, this sets the number of significant figures to use to represent the number in the range 6 (the default) to 15.

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$(), Inkey()

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 includes a vertical bar, the text before the vertical bar is used as the window title.

edit$ The starting value for the text to edit.

maxSz%   Optional, maximum size of the response string.

legal$ An string holding acceptable characters. edit$ 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 upper and lower case characters and integer numbers 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{|}~
```
         The order of extended or accented characters is system and country dependent.

Returns The result is the edited string. A blank string is a possible result.

See also: DlgString(), Input(), Inkey()

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:Chr$(), DelStr$(), LCase$(), Left$(), Len(), Mid$(), Print$(),
        ReadStr(), Right$(), Str$(), Trim(), UCase$(), Val()

Interact()

This function provides a quick and easy way to interact with a user. It displays the interact toolbar at the top of the Spike2 window and pauses the script until a button or a key linked to a button is pressed. Cursors can always be dragged as we assume that they are one of the main ways of interacting with the data. You can limit the user actions when the bar is active.

```
Func Interact(msg$, allow% {,help {, lb1$ {,lb2$ {,lb3$...}}}});
```

msg$ A message to display in the message area of the toolbar. The message area competes with the button area. With many buttons, the text may not be visible.

allow% A number that specifies the actions that the user can and cannot take while interacting with Spike2. 0 allows the user to inspect data and position cursors in a single, unmoveable window. The codes are shown in both decimal and hexadecimal format. The number is the sum of possible activity codes.

1	0x0001	Can change application
2	0x0002	Can change the current window
4	0x0004	Can move and resize windows
8	0x0008	Can use File menu
16	0x0010	Can use Edit menu
32	0x0020	Can use View menu but not ReRun
64	0x0040	Can use Analysis menu
128	0x0080	Can use Cursor menu and add cursors
256	0x0100	Can use Window menu
512	0x0200	Can use Sample menu
1024	0x0400	No changes to y axis
2048	0x0800	No changes to x axis
4096	0x1000	No horizontal cursor channel change

help   This is either the number of a help item (CED internal use) or it is a help context string. This is used to set the help information that is presented when the user presses the F1 key. Set 0 to accept the default help. Set a string as displayed in the Help Index to select a help topic, for example "Cursors: Adding".

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.

If a toolbar created by Toolbar() is present, it is hidden during the Interact() command and restored after Interact() returns.

See also:Toolbar(), ToolbarSet()

Keypress()

This function returns 1 if the `Inkey()` function would return a character, or 0 if it would not. This function and `Inkey()` are provided for compatibility with the MS-DOS version of Spike2 and are not recommended for new scripts.

```
Func Keypress();
```

Returns 1 if a key is ready to read, 0 if there is no key.

See also:`Inkey()`, `Interact()`, `Toolbar()`, `ToolbarSet()`

LastTime()

This function finds the first item on a channel before a time. If a marker filter is applied to the channel, only data in the filter is visible. This function is for time views only.

```
Func LastTime(chan%, time{,&val|code%[]{,data[]|data%[]|&data$}});
```

chan% The channel number in the view to use.

time   The time to search before. Items at the time are ignored. To start a backward search that guarantees to iterate through all items, start at `Maxtime(chan%)+1`.

val   Optional: for waveform channels it returns the waveform value. For event level channels, it returns 0 if the transition is low to high, 1 if the transition is high to low. If there is no event, it returns the level at `time`; 0 for low, 1 for high.

code% This optional parameter is only used if the channel is a marker type (marker, RealMark, TextMark, WaveMark). This is an array with at least four elements that is filled in with the marker codes.

data   Filled with data from RealMark and WaveMark channels. If there is insufficient data to fill it, unused entries are unchanged. An integer array can be used with WaveMark data to collect a copy of the 16-bit data that holds the waveform. If WaveMark data has multiple traces, use `data[points%][traces%]` to get real data and `data%[points%][traces%]` to get the integer data.

data$ A string returned holding the text from a TextMark channel.

Returns The function returns either the time of the next item, or -1 if there are no more items to be found or a negative error code.

See also:`ChanData(), MaxTime(), 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:`Asc(), Chr$(), Str$(), UCase$(), Val()`

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$(), Len(), Mid$(), 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%; n% := Len(arr[][0]); 'get size of first dimension
var m%; m% := Len(arr[0][]); 'get size of second dimension
```

See also:DelStr$(), InStr(), Left$(), Mid$(), Right$(), Str$(), Trim()

LinPred()

Linear prediction can be used to predict future (or past) data values based on a sequence of data. It can also be used to estimate power spectra using the Maximum Entropy or All Poles method. The command generates a set of coefficients that when applied to the previous m points, generate the next predicted point. Some of the explanation for this command relies on technical knowledge; see the references for more information. The command has the following variants:

```
Func LinPred(data[], mMax%{, limit{, out[]{, dir%{, stab%}}}});
Func LinPred(0, out[]{, data[]});                Predict forward
Func LinPred(1, out[]{, data[]});                Predict backwards
Func LinPred(2, stab%);                          Check stability
Func LinPred(3, coef[]);                          Get the coefficients
Func LinPred(4, refl[]);                          Get reflection coefficients
Func LinPred(5, power[], frLo, frHi);         Get estimated power spectrum
Func LinPred(6, poles[][]{, fr[]});           Get poles and frequencies
```

data   An array holding the data to be used to form the linear prediction coefficients or to be used to initialise a prediction based on existing coefficient.

mMax%   The maximum order of the prediction, which is the number of previous points to use to predict each future point. The actual number may be less than this, depending on the value of the limit argument. You will generally want to use the smallest value of m that you can; values in the range 5-50 are common. The value of m must be less than the number of data points and is generally much less. We have set an upper bound of 1024 on mMax% (this is higher than you are likely to need).

limit   The algorithm to calculate the poles is an iterative procedure. The command holds an array of residual values that is initialised to the raw data. Each iteration subtracts data from the residual based on a normalised autocorrelation of the residual (in the range -1 to 1) at increasing lags, aiming to reduce the residual array to a list of zeros. We track a number that models the significance of the remaining data. This number is 1.0 at the start, and is multiplied by $(1-ac^2)$ at each iteration, where ac is the autocorrelation, so the value decreases at each iteration unless ac is 0. If this value becomes less than limit, the iteration will stop. So set limit to 0 for no early stopping, setting a small value greater than 1.0e-31 may stop it early. The actual value of m used is returned by the first command variant.

out   An array of output values predicted by the command. When predicting backwards, the data values are written into the out array so that the first data item in the array is the last predicted point (the oldest). When predicting forwards, the first item in the out array is the first predicted point.

dir%   Optional argument to the setup command that sets the prediction direction. Set 0 or omit for a forward prediction, 1 for reverse.

stab%   The linear prediction coefficients that are generated form a characteristic polynomial, and the roots of this polynomial are the positions of the poles in the z-plane. For the resulting filter to be stable (have an output that does not increase exponentially), the poles must lie inside (corresponding to decaying sinusoids) or on (corresponding to constant amplitude sinusoids) the unit circle. You can use this argument to check the stability of the result and to adjust the pole positions and recompute the coefficients by setting the values:

   0   Do nothing to the poles.

   1   If a pole lies outside the unit circle, reflect it across the unit circle so that a growing sinusoid becomes a decaying sinusoid.

   2   If a pole lies outside the unit circle, move it onto the unit circle, corresponding to a constant sinusoid.

   3   Move all poles onto the unit circle. This produces an output that neither grows nor decays with time.

coef   An array to be filled in with coefficients. The array can be any size, but only the points corresponding with coefficients will be set.

refl   An array to be filled with reflection coefficients. These values represent the proportion of the residual that was subtracted at each iteration.

power   An array to be filled in with estimated power spectrum components.

frLo   A value in the range -0.5 to 0.5, being the fraction of the sampling frequency that the first bin of power will hold the estimated power spectrum for.

frHi   A value in the range -0.5 to 0.5, being the fraction of the sampling frequency that the last bin of power will hold the estimated power spectrum for.

poles   A matrix with the second dimension of size at least 2. `poles[n][0]` is returned holding the real component of the nth pole, and `poles[n][1]` holds the imaginary component.

fr   An array to be filled in with the frequencies that correspond to the pole positions, in the range -0.5 to 0.5 (fraction of the sampling rate).

Returns The function variant return values are described with each variant, or are 0 if not described.

Setup `Func LinPred(data[], mMax%{, limit{, out[]{, dir%{, stab%}}}});`

This command must be used before any other as it calculates the initial set of coefficients. The command returns the number of coefficients that have been generated (this will be the same as `mMax%` if `limit` is 0). The command can be used to generate predicted data, or to set up the system for further `LinPred()` commands. The following example takes 1000 data points from near the start of a waveform channel, predicts the next 100 points and writes them to a memory channel.

```
const chan%:=1;                            'a waveform channel
var data[1000], out[100], sTime, bsz := BinSize(chan%);
var n% := ChanData(chan%, data, 100*bsz, MaxTime(), sTime);
var m% := LinPred(data, 20, 0, out, 0, 0); 'predict forwards
var mc% := MemChan(9, 0, bsz);             'new RealWave channel
MemSetItem(mc%, 0, sTime + n%*bsz, out[]); 'save the data
ChanShow(mc%);                             'display it
```

The predicted data will not be the same as the actual data that follows the first 1000 points unless the first 1100 points are composed of the sum of constant amplitude sinusoids. The command forms a mathematical model of the data held in the data[] array based on the assumption that the spectral components are not changing with time

(the signal is stationary) and that the data is modelled by a set of resonances. Unless you set `stab%` to 3, the result will usually decay with time.

Predict forwards and backwards

```
Func LinPred(0, out[]{, data[]});                    Predict forward
Func LinPred(1, out[]{, data[]});                    Predict backwards
```

These two command versions fill the `out[]` array with data predicted by the coefficients established by the setup version of the command. The setup command `dir%` argument will have set up the command to generate output that joins up with the original data array, or with any output if the setup command generated output. You can choose to continue generating output in the same direction, in which case you must NOT supply the `data[]` array, or you can supply a `data[]` array of at least the size of the number of coefficients to reset the prediction and you can then run forwards or backwards. By supplying a `data[]` array, you are not recalculating the coefficients, these remain unchanged; you are reloading the data points that are used with the coefficients to predict values. If you have m coefficients, when going forwards, the last m data points of `data[]` are used; when going backwards, the first m data points. For example, if we wanted to extend the previous example to predict the 100 data points that might have led up to the original 1000 points we could add the lines:

```
var back[100];
LinPred(1, back, data);                    'predict back from start
MemSetItem(mc%, 0, sTime-100*bsz, back[]);'save the data
```

If we had just used `LinPred(1, back);` this would have caused an error as the previous use of the command was to go forwards. These command variants return 0.

Check stability

```
Func LinPred(2, stab%);                    Check stability
```

This command variant is used to check that the poles of the characteristic polynomial lie within the unit circle. The command returns the distance of the pole furthest from the origin of the z-plane. This should be less than or equal to 1.0 for a stable set of coefficients. The `stab%` argument can be used to adjust the pole positions, as described above. If you are using this command to replace a short stretch of damaged data (like fixing a scratch in a record), you may want to predict both forwards and backwards across the damaged data, then mix the two predictions together. If the data used to generate the coefficients is not stationary, it will decay across the gap, in which case using `stab%` set to 3 may do what you want.

Get coefficients

```
Func LinPred(3, coef[]);                    Get the coefficients
```

This function returns the number of coefficients and returns them in the `coef[]` array. Note that the first coefficient is the one that is multiplied with the most recent data point (when going forward), that is the coefficients run backwards compared to the data.

Get reflection coefficients

```
Func LinPred(4, refl[]);                    Get reflection coefficients
```

This function returns the number of reflection values that are available (the same value as the number of coefficients) and sets `refl` to the results of the auto-correlations done at each lag as the original data was analysed. The results are normalised such that the results lie in the range -1 to 1. The first value is the autocorrelation at a lag of 1 point, the next is for a lag of 2 points, and so on. The term reflection comes from the use of this technique in seismology.

Power spectrum

```
Func LinPred(5, power[], frLo, frHi);          Get estimated power spectrum
```

The variant generates an estimate of the power spectrum based on the representation of the original as the sum of a set of resonances. You must leave `stab%` at 0 when using the command for this purpose. This method is particularly effective if you know that the data contains a small number of constant frequency sinusoids and can allow you to separate sharp peaks that would merge into one using the FFT. However, if your spectrum is not representable as the sum of a set of resonances, the result may be misleading unless you set `mMax%` large enough. Unlike the FFT, where the resolution of the result depends on the number of points in the transform, here the resolution of the result is determined by the number of bins and the frequency range that you set. Make sure you set the bin width small enough so you do not miss a very narrow peak. The frequencies are defined in terms of a fraction of the sampling rates from -0.5 to 0.5 (but note that -0.5 is the same frequency as 0.5); you will find that the power at frequency *f* is equal to the power at frequency *-f*.
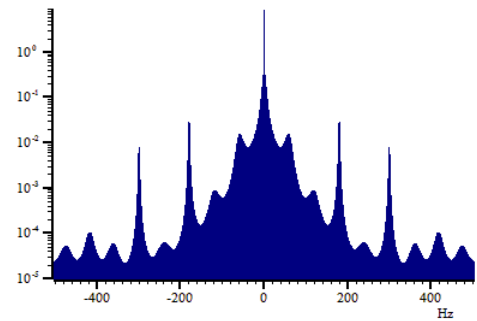
The result is scaled so that the integral of the power from -0.5 up to (but not including) 0.5 is equal to the mean square of the values in the original `data[]` array used in the setup call. For example, we could extend the previous examples to display the power with:

```
const bins% := 10001;                          'Bins in spectrum
var power[bins%];                              'Array to hold power
LinPred(5, power, -0.5, 0.5);
var rv%:=SetResult(bins%,1/(bsz*(bins%-1)),-0.5/bsz,"Power","Hz");
ArrConst([], power);Optimise();WindowVisible(1); ' display result
var MeanSq := ArrDot(data,data)/Len(data);  'Mean square of data
var MemSum := ArrSum(power[:bins%-1])/(bins%-1); 'Power integral
Message("Mean sumSq = %g, SumPower = %g", MeanSq, MemSum);
```

If you try this with synthetic data constructed from sinusoids you will find that `MemSum` < `MeanSq`. The reason is that the purer the sinusoid, the higher and narrow the peak in the power spectrum, so the narrower the bins must be in the power spectrum to capture it.

Get Poles and frequencies   `Func LinPred(6, poles[][]{, fr[]});`   Get poles and frequencies

In the z-plane, the coefficients are represented by a set of poles. You can use this variant to read back the positions of the poles as complex numbers. The poles are sorted in order by their real components, so the pole pairs for each imaginary root should be adjacent. You can also read back the frequencies (as a fraction of the sample rate in the range -0.5 to 0.5) at which the poles are located.

References   Claerbout, Jon F. (1976). "Chapter 7 - Waveform Applications of Least-Squares." Fundamentals of Geophysical Data Processing. Palo Alto: Blackwell Scientific Publications. This has an explanation of the algorithm (due to John P Burg) that we implement.

There is an overview of linear prediction and the Maximum Entropy (All Poles) method of power spectrum estimation in Numerical Recipes, The Art of Scientific Computing, by Press, Flannery, Teukolsky and Vettering.

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.

See also: `ATan()`, `Cos()`, `Exp()`, `Log()`, `Pow()`, `Sin()`, `Sqrt()`, `Tan()`

LnGamma()

The gamma function $\Gamma(x)$ has the useful property that $\Gamma(n+1)$ is the same as $n!$ (*n* factorial) for integral values of *n*. However, it increases very rapidly with *x*, reaching floating-point infinity when *x* is 172.62. To avoid this problem, the script returns the natural logarithm of the gamma function. The definition of the gamma function is:

$$\Gamma(a) = \int_0^\infty t^{a-1}e^{-t}dt$$

```
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.

See also: `GammaP()`, `BinomialC()`

Log()

Takes the logarithm to the base 10 of the argument.

```
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.

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

LogHandle()

This returns the view handle of the log window (which always exists).

```
Func LogHandle();
```

See also: `Print()`, `PrintLog()`, `View()`

MarkEdit()

This changes the data stored in a marker at a particular time. You can get the data using `LastTime()` and `NextTime()`.

```
Func MarkEdit(chan%, time, code%[]{, data$|data[]|data%[]});
```

chan%   The marker, TextMark, WaveMark or RealMark channel to edit.

time   The time of the marker (must match exactly).

code%   Array of 4 marker codes (bottom 8 bits used) to replace markers in the channel.

data   The data to replace the data in the marker. If you use integer data with a WaveMark, the bottom 16 bits of each integer replace the data. To update a

WaveMark with multiple traces use a two-dimensional array, for example with 32 points and 4 traces use `var data[32][4];` to declare the array.

Returns 0 if a marker was edited, or -1 if no marker exists at `time`.

See also:`LastTime(), MarkInfo(), MarkMask(), MarkSet(), NextTime()`

MarkInfo()

This function is used to get information on the extended marker types (TextMark, WaveMark and RealMark).

```
Func MarkInfo(chan% {,&pre% {,trace%}});
```

`chan%` The channel to get the information from.

`pre%` If present, returned as the number of pre-peak points for WaveMark data.

`trace%` If present, returned as the number of traces (electrodes) in the WaveMark data.

Returns For WaveMark data it returns the number of waveform points, for TextMark data it returns the maximum string length and for RealMark data it returns the number of reals attached to each marker. For all other channel types, it returns 0.

See also:`LastTime(), MarkEdit(), MarkMask(), MarkSet(), NextTime()`

MarkMask()

This function sets the mask for a marker, WaveMark, TextMark or RealMark channel. Each data item in one of these channels has four marker codes. Each code has a value from 0 to 255. Each data channel (and duplicated channel) has its own marker filter that determines the visible data items. A marker filter has four masks, one for each of the four marker codes. For each mask, you can specify which codes are wanted. There are two marker filter modes. In the diagrams, a marker data item is represented as a time stamp, four marker codes and data values that depend on the marker type.

Mode 0 (AND) A marker data item is allowed through the mask if each of the four codes in the data item is present in the corresponding mask. We think of this as *and* mode because for the filter to pass the marker, marker code 0 must be in mask 0 *and* marker code 1 in mask 1 *and* marker code 2 in mask 2 *and* marker code 3 in mask 3. In this mode, masks 1, 2 and 3 are usually set to accept all codes and masking is used for layer 0.
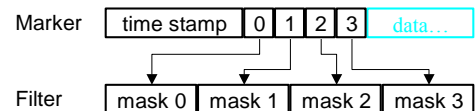
Mode 1 (OR) A marker data item is allowed through the mask if any of the four marker codes is present in mask 0. A code 00 is only accepted for the first of the four marker codes. Masks 1 to 3 are ignored. We think of this as or mode because marker code 0 or 1 or 2 or 3 must be present in mask 0 for the filter to pass the marker for display or analysis. This mode can be used with spike shape data (WaveMark) where two spikes have collided and the marker represents a spike of two different templates.
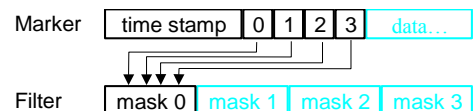
There are three command variants, the first sets the filter codes, the second the filter mode and the third allows you to read and set the mask:

```
Func MarkMask(chan%, layer%, set%, code%|code$ {,code%|code$...});
Func MarkMask(chan%, mode%);
Func MarkMask(chan%, mask%[]{[]}{, write%});
```

`chan%` The channel number to work on. If this is not a suitable channel, the function does nothing and returns a negative error code.

`layer%` The layer of the mask in the range 0 to 3 or -1 for all layers.

set%   1 to include codes in the mask, 0 to exclude codes or -1 to invert the mask. Inverting a mask changes all included codes to excluded and vice versa.

code%   A number in the range 0 to 255 setting a code to include or exclude. You can specify more than one code at a time. -1 is also allowed, meaning all codes.

code$ Each character in the string is converted to its ASCII value, and used as a code.

mode%   The variant with two arguments returns the current mode of the marker filter and optionally sets the mode of matching as 0 or 1 or -1 for no change.

mask%   This is an array that is usually declared as `mask%[256][4]` or `mask%[256]` that maps onto the mask. The second dimension sets the layer. A one-dimensional array maps onto layer 0. When reading, elements are set to 0 or 1. When writing, zero element clear and non-zero elements set the corresponding item in the mask. If the array size does not match 256 by 4, data is transferred between array items that map onto the mask.

write%   If this optional argument is omitted or zero, the `mask%` array is filled with data. If it is 1, the mask data is copied to the mask.

Returns   0 if all is OK, or a negative error code.

A common requirement is to allow all markers to be used. This is achieved by:

```
MarkMask(chan%, -1, 1, -1);  'Set all codes for all layers
```

To fill or empty or invert a complete layer use:

```
MarkMask(chan%, layer%, set%, -1); 'Apply set% to entire layer
```

This example sets the keyboard marker channel mask (channel 31) to show only markers 0 and 1 (start and stop recording markers) and key presses for A, B, C and D:

```
MarkMask(31, 0);                     'set mode 0
MarkMask(31,-1, 1,-1);               'include everything (reset)
MarkMask(31, 0, 0, -1);             'exclude everything in layer 0
MarkMask(31, 0, 1, 0, 1, "ABCD"); 'include the codes we want
```

You can use this command together with `ChanDuplicate()` to split a marker channel into several channels based on marker codes.

See also:`ChanDuplicate()`, `MarkEdit()`, `MarkInfo()`, `MarkSet()`

MarkSet()

This sets the marker codes of a marker, WaveMark, TextMark or RealMark channel in a time range. If a marker filter mask is set, only data that is passed by the mask is changed.

```
Func MarkSet(chan%, sT, eT, code%[]);
Func MarkSet(chan%, sT, eT, c0%{, c1%{, c2%{, c3%}}});
```

chan%   The channel to process.

sT,eT   The time range to process.

code%   An array of 4 integers holding the new marker codes in the range 0 to 255 or -1 for a code that is unchanged.

c0-c3%   One to four marker codes as an alternative to `code%[]`. Use values 0 to 255 to change a code and -1 (or omit the value) to leave a code unchanged.

Returns   The number of markers that were changed.

See also:`LastTime()`, `MarkEdit()`, `MarkMask()`, `MarkShow()`, `NextTime()`

MarkShow()

This sets which of the four marker codes to display for one or more marker channels and the format used when displaying the marker code in a time view. Setting the default draw mode with `DrawMode(chan%,0)` resets the displayed code and drawing mode to 0.

```
Func MarkShow(cSpc{, code%{, mode%}});
```

cSpc   A channel specifier or -1 for all, -2 for visible or -3 for selected channels.

code%   This optional argument sets which of the 4 marker codes to display in the range 0 to 3, or -1 or -2 for no change (-2 returns the current drawing mode).

mode%   Omit or set to -1 for no change. Otherwise set 1 to display all codes as two hexadecimal digits, or 0 to show codes 0x20 to 0x7e as the character equivalent.

Returns The original marker code to display (0-3) or, if `code%` is -2, the drawing mode.

See also:`DrawMode()`, `LastTime()`, `MarkEdit()`, `MarkMask()`, `NextTime()`

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 and/or vectors. In matrix terms, this evaluates $\mathbf{A} = \mathbf{BC}$ where $\mathbf{A}$ is an m rows by n columns matrix, $\mathbf{B}$ is an m by p matrix and $\mathbf{C}$ is a p by n matrix. Vectors of length v are treated as a v by 1 matrix.

```
Proc MATMul(a[]{[]}, b[]{[]}, c[]{[]});
```

a   A m by n matrix of reals or a vector of length m (n is 1) to hold the result.

b   A m by p matrix or a vector of length m (p is 1).

c   A p by n matrix or a vector of length p (n must be 1).

If you pass any of `a`, `b` or `c` as a vector, they are treated as a n by 1 matrix, where n is the length of the vector. Use the `trans()` operator to convert a vector to a 1 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. This procedure physically moves the data, unlike the `trans()` or `` ` `` operator, which remaps the matrix without moving any data. It is usually much more efficient to use `trans()`.

```
Proc MATTrans(mat[][]{, src[][]});
```

mat   A m by n matrix 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 n by m matrix to transpose.

See also:`ArrAdd()`, `MATMul()`

Max()

This function returns the index of the maximum value in an array, or the maximum of several real and/or integer variables.

```
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 array index of the maximum.

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

Maxtime()

In a time view, this returns the maximum time in seconds in the file or in a channel, or the current sample time during sampling. In a result view, it returns the number of bins.

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

chan%   Optional channel number for time views, ignored in result views. If present, the function gets the maximum time in the channel ignoring any marker filter. If there is no data in the channel, the return value is -1.

Returns   The value returned is negative if the channel does not exist. If the current view is of the wrong type the script stops with an error.

To find the time of the last item in the marker filter on a channel with a marker filter set:

```
time := LastTime(chan%, MaxTime(chan%)+1.0);
```

With a marker filter set, Spike2 has to search the data to find a marker that is in the filter. `MaxTime()` returns the last data item in the channel regardless of the marker code. The `+1.0` is because `LastTime()` finds data before the search time.

See also:`Len()`, `LastTime()`, `NextTime()`, `Seconds()`

MeasureChan()

In the current XY view, this function adds or changes a measurement channel created with `MeasureToXY()` using the values set by `MeasureX()` and `MeasureY()`. This command implements some of the functionality of the XY plot setting dialog.

```
Func MeasureChan(chan%, name${, pts%});
```

chan%   This is 0 to add a new channel or the number of an existing channel to change settings. `MeasureToXY()` creates an XY view with one channel, so you will usually call this function with `chan%` set to 1. You can have up to 32 measurement channels in the XY view.

name$ This sets the name of the channel and can be up to 9 characters long.

pts%   Sets the maximum number of points for this channel, if omitted or set to zero then all points are used. When a points limit is set and more points are added, the oldest points are deleted.

Returns   The channel number these `settings` were applied to or a negative error code.

See also:`CursorActive()`, `MeasureToXY()`, `MeasureX()`, `MeasureY()`

MeasureToChan()

This creates a new Event, Marker or RealMark channel with an associated measurement process and cursor 0 iteration method for the current time view. The `Process()` command adds items to the new channel based on active cursor measurements. Before calling this function, use `MeasureX()` to set the measurement method to 102 (Time) and set `expr1$` to generate the time stamps of the new items. If you select any other method this command will return an error code. The iteration must produce times in ascending order unless the output is to a memory channel.

If you are creating a RealMark channel, you must call `MeasureY()` before this function to define the measurement to attach to each data item added to the new channel.

```
Func MeasureToChan(dest%, name$, type%, mode%, chan%, min|exp$
        {, lv|lv${ ,hw{, flgs%{, qu${, width{, lv2|lv2$}}}}}});
```

dest%   This is the output channel number or zero for the lowest numbered, unused memory channel. Set 1 to 400 for channels in the data file and 401 to 700 for specific memory channels. It is a fatal error to set a channel that is in use.

name$ The output channel name. Channel units, if required, are inferred from `chan%` and the measurement method.

type%   This sets the output channel type, as for `ChanKind()`. The allowed types are: 2 or 3 for Event, 5 for a Marker, and 7 for RealMark. If you wish to store data other than event times in the file, you should set `type%` to 7.

mode%   This is the cursor 0 iteration mode. Modes are the same as in `CursorActive()` but not all modes can be used. Valid modes are:

4 Peak find	8 Falling threshold	14 Data points	19 Outside levels
5 Trough find	12 Slope peak	16 Expression	20 Inside levels
7 Rising threshold	13 Slope trough	17 Turning point	

chan%   This is the channel searched by the cursor 0 iterator. In expression mode (16), this is ignored and should be set to 0.

min   This is the minimum allowed step for cursor 0 used in all modes except 16.

exp$ This is the expression that is evaluated in mode 16.

lv   This number or string expression sets the threshold level for threshold modes and the peak size for peak and trough modes. It is in the y axis units of channel `chan%` or y axis units per second for modes 12 and 13. In mode 14, it sets the points as a number and defaults to 1. This argument is ignored and should be 0 or omitted for modes that do not require it.

hw   This sets the hysteresis in y axis units for threshold modes. This argument is ignored and should be 0 or omitted for modes that do not require it. For backwards compatibility, if `width` is omitted, it sets the width in seconds for slope measurements.

flgs%   This is the sum of option flags. Add 1 to force a common x axis Add 2 for user checks on the cursor positions. The default value is zero.

qu$ This sets the qualification expression for the iteration. If left blank then all iteration positions will be used. If not blank, and it evaluates to non-zero, then the iteration is skipped.

width   This sets the width, in seconds, for slope measurements. Set this 0 or omit it in modes that do not require it.

lv2   This number or string expression is used with `lv` to set the two threshold levels for cursor iteration modes 19 and 20.

Returns The function result is the number of the created channel.

See also:`CursorActive()`, `MeasureChan()`, `MeasureX()`, `MeasureY()`, `Process()`

MeasureToXY()

This creates a new XY view with a measurement process and cursor 0 iteration method for channels in the current time view. It creates one output channel with a default measurement method. Use `MeasureX()`, `MeasureY()` and `MeasureChan()` to edit the method and add channels. Use `Process()` to generate the plot. The new XY view is the current view and is invisible. Use `WindowVisible(1)` to make it visible. These commands implement the functionality of the Measurements to XY view dialog.

```
Func MeasureToXY(mode%, chan%, min|exp$
          {, lv|lv${ ,hw{, flgs%{, qu${, width{, lv2|lv2$}}}}}});
```

mode%   This is the cursor 0 iteration mode. Modes are the same as in `CursorActive()` but not all modes can be used. Valid modes are:

4 Peak find	8 Falling threshold	14 Data points	19 Outside levels
5 Trough find	12 Slope peak	16 Expression	20 Inside levels
7 Rising threshold	13 Slope trough	17 Turning point	

chan%   This is the channel searched by the cursor 0 iterator. In expression mode (16), this is ignored and should be set to 0.

min   This is the minimum allowed step for cursor 0 used in all modes except 16.

exp$ This is the expression that is evaluated in mode 16.

lv   This number or string expression sets the threshold level for threshold modes and the peak size for peak and trough modes. It is in the y axis units of channel `chan%` or y axis units per second for modes 12 and 13. In mode 14, it sets the points as a number and defaults to 1. This argument is ignored and should be 0 or omitted for modes that do not require it.

hw   This sets the hysteresis in y axis units for threshold modes. This argument is ignored and should be 0 or omitted for modes that do not require it. For backwards compatibility, if `width` is omitted, it sets the width in seconds for slope measurements.

flgs%   This is the sum of option flags. Add 1 to force a common x axis. Add 2 for user checks on the cursor positions. The default value is zero.

qu$ This sets the qualification expression for the iteration. If left blank then all iteration positions will be used. If not blank, and it evaluates to non-zero, then the iteration is skipped.

width   Set this value in seconds; use 0 or omit it in modes that do not require it. For slopes, it sets the time over which the slope is measured. For peaks and troughs,

it sets the maximum peak width (use 0 for no maximum). For threshold modes 7, 8, 19 and 20, it sets the minimum crossing time (**Delay** in the dialog).

lv2   This number or string expression together with lv sets the two threshold levels for cursor iteration modes 19 and 20.

Returns   The function result is an XY view handle or a negative error code.

Arguments passed as strings are not evaluated until data is processed. Invalid strings generate invalid measurements and no data points in the XY view.

Example   This generates a plot of peak values in channel 1 of the current time view. Peaks must be at least 0.1 seconds apart and the data must fall by at least 1 y axis unit after each peak.

```
var xy%;                          'Handle of new xy view
xy%:=MeasureToXY(4, 1, 0.1, 1);   'Peak, chan 1, min step 0.1, amp 1
WindowVisible(1);                 'Window is invisible, so show it
MeasureX(102, 0, "Cursor(0)");    'x = Time, no channel, at cursor 0
MeasureY(100, 1, "Cursor(0)");    'y = Value of chan 1 at cursor 0
MeasureChan(1, "Peaks", 0);       'Set the title, no point limit
Process(0.0, View(-1).MaxTime(),0,1); 'Process all the data
```

See also:CursorActive(), MeasureChan(), MeasureX(), MeasureY(), Process()

MeasureX()

MeasureX() and MeasureY() set the x and y part of a measurement. The settings are saved but have no effect until MeasureChan() or MeasureToChan() are used to change or create a channel. This command implements some of the functionality of the XY plot setting dialog. The current view must be the target of the measurements.

```
Func MeasureX(type%, chan%, expr1|coef% {,expr2 {,width}});
Func MeasureY(type%, chan%, expr1|coef% {,expr2 {,width}});
```

type%   This sets the x or y measurement type. Values less than 100 match those for the ChanMeasure() command (except types 5 and 1 are identical). When using MeasureX() with MeasureToChan(), the only valid type% is 102.

1 Area	5 Area (scaled)	9 Minimum	13 Abs max.	17 Mean in X
2 Mean	6 Curve area	10 Peak to Peak	14 Peak	18 SD in X
3 Slope	7 Modulus	11 RMS Amplitude	15 Trough	19 Mean of abs
4 Sum	8 Maximum	12 Standard deviation	16 RMS error	

Values from 100 up are:

100 Value	104 0-based fit coefficient	108 Value product
101 Value difference	105 User entered value	109 Value less baseline
102 Time	106 Expression	
103 Time difference	107 Value ratio	

chan%   This is the channel number for the measurement. For time, user entered and expression measurements it is ignored and should be set to 0.

expr1   Either a real value or a string expression that sets the start time for measurements over a time range, the position for time (102) and value measurements and the expression used for measurement type 106.

coef%   The zero-based coefficient number for measurement type 104.

expr2   Either a real value or a string expression that sets the end time for measurements over a time range and the reference time for single-point measurements and differences. Set an empty string when width is required and this is not.

width   This is the measurement width for value and value difference measurements. The default value is zero.

Returns   The function return value is zero or a negative error code.

See also:CursorActive(), MeasureChan(), MeasureToChan(), MeasureToXY()

MeasureY()

This is identical to `MeasureX()` and sets the y part of a measurement for a measurement channel. The settings are saved but have no effect until `MeasureChan()` is used to change or create a channel. See the `MeasureX()` documentation for details.

```
Func MeasureY(type%, chan%, expr1$ {,expr2$ {,width}});
```

See also:`MeasureX()`

MemChan()

This function creates a new channel in memory and attaches it to the file in the current time view. You can have up to 300 memory channels per file. There are two variants: the first lets you specify the channels settings, the second (added at version 6.03) copies settings from an existing channel.

```
Func MemChan(type%{, szLev%{, binsz{, pre%{, trace%}}}});
Func MemChan(0, copy%);
```

type% The type of channel to create. Codes are:
      1 Waveform   4 Level (Event+-)   7 RealMark
      2 Event (Event-)   5 Marker   8 TextMark
      3 Event (Event+)   6 WaveMark   9 Real wave

szLev% For TextMark, RealMark and WaveMark types, sets how many characters, reals or waveform points to attach to each item. In Level channels, omit or set 0 for a low initial level, 1 for high. Set 0 or omit for other channels.

binsz Used for waveform and WaveMark data to specify the time interval between the waveform points. This is rounded to the nearest multiple of the underlying time resolution. If you set this 0 or negative, the smallest bin size possible is set.

pre% This must be present for WaveMark data to set the number of pre-trigger points.

trace% Optional, default 1, sets the number of WaveMark traces in range 1 to 4.

copy% A channel from which to copy settings, including the channel comment, title, units and scale factors. The channel must exist.

Returns The new channel number, 0 if there is no free channel, or a negative error code.

Channels created by the first variant have default titles, units, comments and scale factors. You can set these with `ChanTitle$()`, `ChanUnits$()`, `ChanComment$()`, `ChanScale()` and `ChanOffset()`. This example copies a channel of any type:

```
func CopyWave%(chan%)
var mc%;
mc% := MemChan(0, chan%);                    'Create waveform channel
if mc%>0 then                                'Created OK?
   ChanComment$(mc%, "Copy of channel "+Str$(chan%));
   MemImport(mc%, chan%, 0, MaxTime());  'Copy data
   ChanShow(mc%);                            'display new channel
endif;
return mc%; 'Return the new memory channel number
end;
```

See also:`ChanNew()`,`MemImport()`,`MemSave()`,`MemSetItem()`

MemDeleteItem()

This function deletes one or more items from a channel created by `MemChan()`. To delete the entire channel use `ChanDelete()`.

```
Func MemDeleteItem(chan% {,index% {,num%}});
```

chan% The channel number of a channel created by `MemChan()`.

index% The ordinal index into the channel to the item to delete. The first item is numbered 1. If you specify index -1 or omit this argument, all items are deleted.

num%   The number of items to delete from index%. Default value is 1.

Returns The number of items deleted or a negative error code.

See also: MemDeleteTime()

MemDeleteTime()

This function deletes one or more items from a memory channel based on a time range. If a marker filter is set, only items in the filter are deleted unless you add 4 to mode%.

```
Func MemDeleteTime(chan%, mode%, t1 {,t2});
```

chan%   The channel number of a channel created by MemChan().

mode%   This sets the items to delete and how to interpret the time range t1 and t2:
  0   A single item is deleted. t1 is the item time and t2 is a timing tolerance (0 if t2 is omitted). The nearest item to t1 in the time range t1-t2 to t1+t2 is deleted. If there are no items in the range, nothing is deleted.
  1   Delete all items from time t1-t2 to t1+t2. If t2 is omitted, 0 is used.
  2   Delete the first item from time t1 to t2. If t2 is omitted it is taken as t1.
  3   Delete all items from time t1 to t2. If t2 is omitted, it is taken as t1.
  +4 If you add 4 to the mode, any marker filter for the channel is ignored.

t1, t2   Two times, in seconds, which set the time range for items to delete.

Returns The number of items deleted, or a negative error code.

See also: MemChan(), MemDeleteItem(), MemGetItem(), MemSetItem()

MemGetItem()

This returns information about a memory channel item. The items are identified by their ordinal position in the channel, not by time, and any mask set for markers is ignored.

```
Func MemGetItem(chan% {,index% {,code%[] {,&data$|data[]}}});
Func MemGetItem(chan%, index%, &wave|&wave%|wave[]|wave%[]{,&n%});
```

chan%   The channel number of a channel created by MemChan().

index%   The ordinal index into the channel to the item required. The first item is numbered 1. If you omit the index, or specify index 0, the function returns the number of items in the channel.

The remaining fields are only allowed if the index is non-zero.

code%   This is an integer array of at least 4 elements that is returned holding the channel marker codes. If there are no markers, the codes are all set to 0.

data   This must be a string variable for TextMark data or a real array for RealMark or WaveMark data. It is returned holding the data from the item. If the data type is incorrect for the channel, it is not changed. For an array, the point count returned is the smaller of the size of the array and the number of values in the item. You can use a two dimensional array to collect WaveMark data with multiple traces. The second dimension is for the traces: var data[points%][traces%];

wave   This argument collects waveforms from waveform channels. If a real variable or array is passed, the waveform data is in user units. If an integer variable or array is passed, the data is a copy of the 16-bit integer data used by Spike2 to store waveforms. The maximum number of elements copied is the array size.

         When an array is used, only contiguous data is returned. A gap in the data (when the interval between two points is greater that the sample interval for the channel) terminates the data transfer.

         You may find that ChanData() is easier to use when you want to collect waveform (or event) data based on a time range.

n%   If the previous argument is an array this optional argument returns the number of data items copied into the array.

Returns   For index% of 0 or omitted, the function returns the number of items in the channel. If an index is given that is outside the range of items present, the function returns -1. Otherwise, it returns the time of the item.

See also:ChanData(), MemImport(),MemSave(),MemSetItem()

MemImport()

This function imports data into a channel created by MemChan(). There are some restrictions on the type of data channel that you can import from, depending on the type of the destination channel.

Destination	Source	Restrictions
Waveform	Waveform	Data copied, but must match sample interval
	All others	Not available, see EventToWaveform()
Event	Waveform	Can extract event times
	All others	Times are extracted from the channel
Level	Waveform	Can extract event times
	All others	Times are extracted from the channel. First time in destination is assumed to be a low to high transition.
Marker	Waveform	Can extract event times, coded for peak/trough etc.
	Event	Marker codes all set to 0
	Level	Marker codes set 00 for falling edge, 01 for rising
	All others	Marker codes are copied
TextMark	Waveform	Can extract event times, coded for peak/trough etc.
	Event	Copies times, marker codes set 0, empty strings
	Level	As Event except falling edge coded 00, rising 01
	TextMark	Copies all data, strings may be truncated if too long.
	All others	Copies marker information, empty strings
RealMark	Waveform	Can extract event times, coded for peak/trough etc.
	Event	Times copied, marker codes and reals set to 0
	Level	As Event except falling edge coded 00, rising 01
	RealMark	Copied, real data truncated or zero padded as needed
	WaveMark	Copied, waveform to reals, padded/truncated
	All others	Marker portion copied, reals filled with 0
WaveMark	Waveform	Special option, event channel marks waveforms
	Event	Copies times, marker codes set to 0, waveform set 0
	Level	As Event except falling edge coded 00, rising 01
	WaveMark	Copies all data, waveforms aligned on trigger point
	All others	Copies marker, waveform filled with zeros

You can extract events or markers from waveform data using peak search and level crossing techniques. You can convert waveform data to WaveMark data with a special option that chops sections of waveform data out based on event times on a third channel.

Import compatible channel

The first command variant imports data from a compatible channel. All event and marker channels can be copied to each other, but the information transferred is the lowest common denominator of the two channel types. Missing data is padded with zeros. Waveform data is compatible with itself if both channels have the same sampling rate.

```
Func MemImport(chan%, inCh%, start, end);
```

chan%   The channel number of a channel created by MemChan().

inCh%   The channel number to import data from.

start   The start time to collect data from.

end   The end time to collect data up to (and including).

Extracting events from waveforms

The `mode%`, `time`, `level` and `code%` arguments are used when extracting events from waveform data. The level crossing modes use linear interpolation between points to find the exact time of the waveform crossing. The peak and trough modes fit a parabola to the three points around the peak or trough to estimate the time more accurately.

When extracting events to a WaveMark channel from waveform data, the time saved is the time of the start of the waveform section, not the peak/trough or level crossing time. The saved waveform data starts the number of points before each event set by the `pre%` parameter to `MemChan()`. The WaveMark time is adjusted to match the waveform. If both channels do not have the same sampling rate, the copied waveform is set to zero.

```
Func MemImport(chan%,inCh%,start,end{,mode%,time,level{,code%}});
```

`mode%`   The mode of data extraction. The modes are:

> 0   Extract events based on the time of a peak in the waveform. If the destination is a marker, these events are coded as 2 unless `code%` is set.
>
> 1   Extract events based on the time of a trough in the waveform. If the destination is a marker, the events are coded 3 unless `code%` is set.
>
> 2   Extract events based on times when the waveform rises through `level`. If the destination is a marker, the events are coded 4 unless `code%` is set.
>
> 3   Extract events based on times when the waveform falls through `level`. If the destination is a marker, the events are coded 5 unless `code%` is set.

`time`   The minimum time between detected events. Use this to filter noisy signals.

`level`   In modes 0 and 1, this is the distance that the waveform must fall after a peak or rise after a trough. In modes 2 and 3, it is the level to cross to detect an event.

`code%`   If present and positive it overrides the codes based on `mode%` that are applied to the events. The low byte of `code%` sets the first marker code; the remaining marker codes are always 0.

WaveMark from events and waveform data

The special mode to convert waveform to WaveMark data uses an extra channel to mark the waveform sections to be extracted. The waveform must have the same sampling rate as set for the WaveMark channel. The saved waveform data starts the number of points before each event set by the `pre%` parameter to `MemChan()`.

```
Func MemImport(chan%, inCh%, start, end, eCh%);
```

`eCh%`   A channel holding event times to mark the waveform sections to extract. The time saved is the time of the first point in each waveform section. If the channel contains marker codes, these are copied to the memory channel.

Returns It returns the number of items added to the channel, or a negative error code.

See also:`MemChan()`, `MemGetItem()`, `MemSave()`, `MemSetItem()`

MemSave()

This writes a channel created by `MemChan()` to the data file associated with the current window, making the data permanent. The memory channel is not changed; use `ChanDelete()` to remove it.

```
Func MemSave(chan%, dest%{, type%{, query%{, bufSz%}}});
```

`chan%`   A channel created by the `MemChan()` function.

`dest%`   The destination channel in the file. This must be in the range 1 to the maximum number of channels allowed in the data file.

`type%`   The type of data to save the data as. The type selected must be compatible with the data in the memory channel. Codes are:

0 Same type (default)	3 Event (Evt+)	6 WaveMark	9 RealWave
1 Waveform	4 Level (Evt+-)	7 RealMark	
2 Event (Evt-)	5 Marker	8 TextMark	

The special code -1 means append the memory channel to an existing channel. The new data must occur after the last item in the `dest%` channel and the `dest%` channel must be of a compatible type to the memory channel.

query% If this is not present or zero, and the `dest%` channel is already in use, the user is queried about overwriting it. If this is non-zero, no query is made.

bufSz% If present and greater than 0, this sets the disk block size in bytes for waveform and RealWave data. If omitted (recommended) or zero, a default size is used.

Returns The number of items written, or a negative error code.

See also: ChanDelete(), ChanWriteWave(), MemChan(), MemImport()

MemSetItem()

This function edits or adds an item in a channel created by `MemChan()`. The item is identified by its ordinal position in the channel and any mask set for markers is ignored.

```
Func MemSetItem(chan%, index%, time{, code%[]{, data$|data[]}});
Func MemSetItem(chan%, index%, time, wave|wave%|wave[]|wave%[]);
```

chan% The channel number of a channel created by `MemChan()`.

index% The index of the item to edit. The first item is number 1. An index of 0 adds a new item to the buffer at a position set by `time` (which must be positive).

time The item time, or -1 for no change. If `index%` is 0, you must supply a time. For a waveform channel, it sets the time of the first data point but if there is already data in the channel, `time` is adjusted by up to half the sampling interval to be compatible with the sampling interval of the channel and the existing data.

code% An integer array of 4 or more elements holding marker codes. From version 7.10, it can be an integer holding $c0+256*(c1+256*(c2+256*c3))$ with `cn` in the range 0-255. If omitted for a marker channel, the codes are set to 0.

data A string for TextMark data or a real array for RealMark or WaveMark, holding the item data. If the data type is incorrect, it is ignored. The number of points or characters set is the smaller of the number passed and the number expected. For RealMark and WaveMark data, if the array is too short, the extra values are unchanged when `index% > 0` (editing) and have the value 0 if `index%` is 0.

WaveMark and waveform real values are limited to the range -5*`scale`+`offs` to 4.99985*`scale`+`offs`. A two dimensional array is allowed for WaveMark data: `var data[points%][traces%];` passed in as `data[][]`.

wave For a waveform you can set one value, or an array. Real values are limited as described above. For integers, the lower 16-bits of the 32-bit integer are copied to the channel (values greater than 32767 or less than -32768 will overflow).

Returns The function returns the index at which the data was stored. If an index is given that is outside the range of items present, the function returns -1.

See also: ChanWriteWave(), MemChan(), MemGetItem(), MemImport(), MemSave()

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 includes a vertical bar, the text before the vertical bar is used as the window title.

arg1,2 The arguments used to replace `%d`, `%f` and `%s` type formats.

You can split the message into multiple lines by including `\n` in the `form$` string. Long messages are truncated.

See also:`Print()`, `Input()`, `Query()`, `DlgCreate()`

Mid$()

This function returns a sub-string 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, there is no limit on the number.

Returns The sub-string. If `index%` is larger than `Len(text$)`, the string is empty.

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

Min()

This function returns the index of the minimum value in an array, or the minimum of several real and/or integer variables.

```
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 array index of the minimum.

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

Minmax()

`Minmax()` finds the minimum and maximum values for result views and time view channels with a y axis, or the minimum and maximum intervals for an event or marker channel drawn as dots or lines. `Min()` and `Max()` are preferred in result views.

```
Func Minmax(chan%, start, finish, &min, &max,{,&minP{,&maxP
                      {,mode% {,binsz {,trig%|edge%{, as%}}}}}}});
```

`chan%`   The channel number in the time or result view.

`start`   The start position in time for a time view, in bins for a result view.

`finish` The end position in time for a time view, in bins for a result view.

`min`   The minimum value is returned in this variable.

`max`   The maximum value is returned in this variable.

`minP`   The position of the minimum is returned in this variable.

`maxP`   The position of the maximum is returned in this variable.

`mode%`   If present, this and the following arguments sets the display mode in which to read the value from a time view. If `mode%` is inappropriate or absent, the current display mode is used. See `DrawMode()` for details. This parameter is ignored in a result view.

Returns Zero if all was well or a negative error code.

See also:`DrawMode()`, `Min()`, `Max()`, `XYRange()`

MMAudio()

This returns information about the audio content of the current multimedia window.

```
Func MMAudio({&rate});
```

rate   If present, this is returned as the sample rate per channel, in Hz.

Returns   The number of audio channels or 0 if no audio or not a multimedia window.

See also:MMOpen(), MMImage(), MMPosition(), MMVideo()

MMFrame()

This command returns a list of real frame times in the current multimedia view.

```
Func MMFrame(tStart, tEnd{, secs[]{, &more%}});
```

tStart   The start of the time range to search for video frames.

tEnd   The end of the time range to search.

secs[]   If present, an array to fill with frame times in seconds.

more%   If present, returned as 1 if the time range holds more frames than can be returned in secs[], returned as 0 if there are no more frames.

Returns   If secs is present, the number of frame times returned; if absent, the number of frames in the time range.

See also:MMOpen(), MMImage(), MMOffset(), MMPosition(), MMVideo()

MMImage()

Copy the image in the current multimedia view to arrays. The two-dimensional arrays that collect the image are treated as x[v][h], where v is a vertical co-ordinate and h is a horizontal co-ordinate. x[0][0] is at the bottom left of the image. Use MMVideo() to get the image size. Sections of the array that do not map onto the image are not changed.

```
Func MMImage(rgb%[][]);
Func MMImage(map[][]|mode%{, r[][]{, g[][]{, b[][]}}});
Func MMImage(-1, h[][]{, l[][]{, s[][]}}});
```

rgb%   If the first argument is a two-dimensional integer array, it is returned with each array element holding one pixel in RGB32 format. Bits 0-7 hold the Red intensity, bits 8-15 hold the Green and bits 16-23 hold the Blue. Bits 24-31 are 0. Any additional arguments are ignored.

mode%   If the first argument is 1, the r,g and b arrays are filled with Red, Green and Blue intensities. If mode% is set to 2, the arrays are filled with Cyan, Magenta and Yellow intensities. The colour components will be in the range 0 to 1.0, inclusive. If mode% is 0, the r array is filled with a monochrome representation of the data with 0 representing black and 1.0 representing white; the g and b arrays are set to 0, if they are present.

If the first argument is -1, the h, l and s arrays return HLS (Hue, Lightness and Saturation) data. Hue is in degrees around a colour hexagon with Blue at 0, Red at 120 and Green at 240 degrees. Lightness represents the amount of white in a colour, from 0 (black) to 1.0 (white). Saturation is a measure of the purity of a colour, from 0 (grey) to 1.0, which is a pure colour.
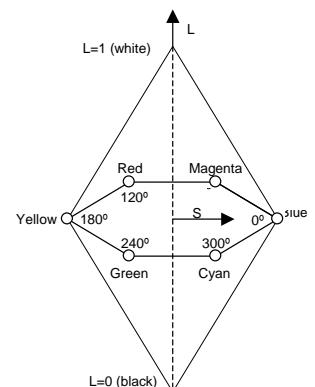
map   If this is present, it should be a 3x3 array that maps the red, green and blue (RGB) in the image into the r, g and b arrays that you provide. The first array index sets the array: 0 for r, 1 for g and 2 for b. The second array index sets the

colour: 0 for red, 1 for green and 2 for blue. The array values set how much of each colour should be added into the result.

If R, G and B (all in the range 0.0 to 1.0), represent the colour of a pixel, the `r` array pixel is set to `R*map[0][0]+G[]*map[0][1]+B*map[0][2]`. The `g` and `b` arrays (if present) are calculated with the first index of `map` set to 1 and 2.

r,g,b   These arrays, when present are filled with RGB colour information. If `map%` is used, the values in the arrays depend on the map values. If `mode%` is used, the values will be in the range 0 to 1.

h,l,s   These arrays are used when the first argument is -1. They are returned holding the hue (0 to 360), lightness (0 to 1.0) and saturation (0 to 1.0).

Returns 1 if the copy succeeded and 0 if it failed.

See also:`EditCopy(), FileSaveAs(), MMFrame(), MMPosition(), MMVideo()`

MMOffset()

You can set the start time of the AVI file in the associated time view. We store this offset in the AVI file header in a supposedly unused region so that you only need do this once per file. Setting a 0 offset restores the AVI header to a standard state.

```
Func MMOffset({offset});
```

offset If present, this sets the new offset in seconds. The offset is stored internally to millisecond resolution. The offset is saved to the AVI file when the multimedia window closes as long as the file is not read-only.

Returns The original multimedia offset at the time of the call, in seconds.

See also: `MMAudio(), MMFrame(), MMImage(), MMPosition(), MMVideo()`

MMOpen()

This counts, opens and returns the view handles of multimedia windows associated with the current time or multimedia view. Multimedia files associated with `fname.smr` are named `fname-1.avi`, `fName-2.avi` and so on. If a multimedia window is opened or located, it becomes the current view. If multiple windows open, the last-opened window becomes the current view. This does not access the `s2video` application window.

```
Func MMOpen({n%{, mode%}});
```

n%   Set to 0 to open or count all associated multimedia files. Set to n (greater than 0) for the nth multimedia file. It is not an error to open a file that is already open.

mode%   This optional argument has the default value 0 and is the sum of the following:
  1   Make new windows visible. Omitting this does not hide an existing window.
  2   Reserved.
  4   Do not open any new windows, do not change the current view. Use this to find a view handle or count the multimedia windows.

Returns If `n%` is 0 or omitted, the return value is the number of associated multimedia files that are open. If `n%` is greater than 0, and a file was opened, or was already open, the return value is the view handle.

See also:`FileClose(), MMAudio(), MMImage(), MMPosition(), MMVideo()`

MMPosition()

This sets and gets the play position and state of the current multimedia window or sets the state of all multimedia windows associated with the current time view. Multimedia windows track any change in the associated time view position. Use this command to

change the play position or state independently of the time view. Use `Rerun()` to replay a time view and all associated multimedia windows together. Multimedia windows play in real time or at the rate set by the last `Rerun()` to the window.

```
Func MMPosition({pos{, sPlay%{, &gPlay%}}});
```

pos   The time in seconds in the multimedia stream to move to. Omit or set it negative for no position change. pos is ignored if `sPlay%` is 2, if 3 pos is a frame offset.

sPlay% This argument controls the play state. Omit it or set it to -1 for no change in the play state. Set 0 to stop playing, 1 to start playing, 2 to track cursor 0, 3 to move by 1 (pos > 0), 0 (pos =0) or -1 (pos < 0) video frames.

gPlay% If this argument is present it is returned holding the play state after the command has run as 0 for not playing and 1 for playing.

Returns The return value is the play position after any changes made by the command have been made or -1 if an error occurred or no window was found. If the command is applied to a time view, the return value is for the lowest numbered multimedia window found.

See also: `MMAudio()`, `MMImage()`, `MMOpen()`, `MMVideo()`, `Rerun()`

MMRate()

This requests the attached `s2video` applications to change the video frame rate. You cannot run faster than the camera frame rate. The rate is achieved by dropping frames from the video stream; the result may be faster or slower than you request.

```
Func MMRate(fps);
```

fps   The desired frames per second. Rates of 0 or less set the minimum of the camera rate or the maximum frame rate set in the `s2video` application.

Returns The number of external windows that this request was sent to.

See also: `MMAudio()`, `MMImage()`, `MMOpen()`, `MMPosition()`, `MMVideo()`

MMVideo()

This returns information about the video content of the current multimedia window.

```
Func MMVideo({&hPix%{, &vPix%{, &fps}}});
```

hPix%   If present, returned as the number of horizontal pixels in the image.

VPix%   If present, returned as the number of vertical pixels in the image.

fps   If present, returned as the frames per second value held in the image file.

Returns 0 if no video is present or this is not a multimedia window or 1 if there is video in the multimedia window.

See also: `MMAudio()`, `MMImage()`, `MMOpen()`, `MMPosition()`

Modified()

This command lets you get (and in some cases, set) the modified state of a view and if it is read only. Beware that clearing the modified flag for views that support this will allow you to close the view without being prompted to save changes.

```
Func Modified({what%{, new%}});
```

what%   0 (or omitted) to get or set the modified state, 1 to get or set the read only state.

new%   -1 (or omitted) for no change, 0 to clear the state, 1 to set the state.

Returns The state at the time of the call, before any change.

The meaning and effect of this routine depends on the type of the current view.

Text view   A text view is considered modified if there are undoable changes since the last save point; such changes will be interactive edits and typing. Changes made by a script do not count as modifications and are not undoable. `Modified(0,0)` makes the current state the last save point without saving; you cannot set the modified flag with this command.

Text views (but not the Log view) can be set read only with `Modified(1,1)` and writable with `Modified(1,0)`. Output sequence and script files open read only if they are read only on disk. This command does not change the read only status of the disk file.

Time view   A Time window counts as modified if you make a change to it that would be written to the underlying `.smr` file. Changes made to memory, duplicate or virtual channels do not count as modifications. Some changes are written immediately, others are buffered up and are only written when the file is closed. You can force the file to commit changes held in buffers and to update the file header to disk with `Modified(0,0)`. Committing changes does not clear the modified flag.

`Modified(1)` reports the read only state. You cannot change the read only state of a time view; it depends on the read only state of the underlying `.smr` file.

Result and XY views   A result or XY view is modified if changed since the last save. You can set and clear the modified flag using `Modified(0, new%)`. There is currently no concept of a read only state for a result or XY views and `Modified(1)` always returns 0.

Other view types   The command is not implemented for other view types and will return 0.

MousePointer()

This command loads permanent mouse pointers from external files, creates new temporary mouse pointers and can delete temporary mouse pointers when they are no longer needed. Spike2 maintains a list of standard mouse pointers (see `ToolbarMouse()` for the list); any pointers added by this function are added to this list and are available for use by the mouse down, move and up routines associated with `ToolbarMouse()`.

```
Func MousePointer(text$|nDel%);
```

text$ This is a text string that defines a new mouse pointer. This is either the path to a data file that holds a cursor or an animated cursor, ending in ".cur" or ".ani" (not case sensitive) or it is a text string that defines a monochrome mouse pointer, as described below. The function returns the number of the new mouse pointer, or 0 if it was not created. Cursors loaded from a file are permanent and exist until Spike2 closes. There is a limit on the number of cursors that Spike2 will manage (currently set to 60); this should be more than enough for any reasonable purpose.

nDel%   The number of a mouse pointer to delete, or -1 to delete all user-defined mouse pointers. You can only delete temporary mouse pointers, and you cannot delete a mouse pointer that is currently in use.

Returns   The number of a newly created mouse pointer or 0 if it was not created or the number of mouse pointers that were deleted.

Text format to create a new pointer   Mouse pointers defined by this function are a 32 x 32 pixel image. Inside this image is the hot spot, being the actual position where the mouse is deemed to be pointing. Each pixel in the image can be screen coloured, black, white or the inverse of the screen. In addition, you can designate a pixel to be the hot spot. This is coded in text as follows:

Cursors are defined by pixel by pixel, starting at the top left, moving horizontally and starting a new row every 32 characters. However, most

Character	Screen	Black	White	Inverse
Normal	space	b	w	i
Hot-spot	h	B	W	I

mouse pointers are much smaller than 32 x 32 pixels, so you can stop a line early by adding a vertical bar character "|" or by a line feed character "\n". Any character that is not a space, b, B, w, W, h, i, I, | or line feed is treated as a space.

You do not need to provide 32 rows; any undefined pixels are treated as if they were filled with spaces. It is not an error to include more than one hot-spot character; the hot-spot position is set by the last hot-spot character in the string.

The example to the right creates a simple mouse pointer that has a 7 x 7 pixel black square around a 5 x 5 white square and the hot-spot in the centre. Although the text could be written on a single line, the form of the mouse pointer is much clearer when it is shown in the code in the same format as it is displayed.

```
var mp% :=
MousePointer(
"bbbbbbb|"
"bwwwwwb|"
"bw   wb|"
"bw h wb|"
"bw   wb|"
"bwwwwwb|"
"bbbbbbb");
```

Loading mouse pointers from files   Cursors created with text strings are monochrome and not animated. You can also load coloured (sometimes called 3D) cursors and animated cursors from files. if you want to experiment with this, you can find suitable files in the WINDOWS\Cursors folder. On my machine, the following loads an animated stopwatch cursor:

```
var ani% := MousePointer("c:\\WINDOWS\\Cursors\\stopwtch.ani");
```

See also:Toolbar(), ToolbarMouse()

MoveBy()

This moves the text caret in a text window relative to the current position by lines and/or a character offset. You can extend or cancel the current selection.

```
Func MoveBy(sel%{, char%{, line%}});
```

sel%   With char% present, if sel% is zero, all selections are cleared. If non-zero the selection is extended to the destination of the move. With char% omitted sel%=0 returns the character offset, 1 the line number and 2 the column.

char%   If line% is absent, the new position is obtained by adding char% to the current character offset in the file. You cannot move the caret beyond the existing text.

line%   If present it specifies a line offset. The new line is the current line number plus line% and the new character position is the current character position in the line plus char%. The new line number is limited to the existing text. If the new character position is beyond the start or end of the line, it is limited to the line.

Returns   It returns the new position. MoveBy(1,0) returns the current position without changing the selection. See sel% (above) to get line and column numbers.

See also:MoveTo(), Selection$()

MoveTo()

This moves the text caret in a text window. You position the caret by lines and/or a character offset. You can extend or cancel the text selection. The first line is number 1.

```
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%   If line% is absent, this sets the new position in the file. You cannot move the caret beyond the existing text. 0 places the caret at the start of the text.

line%   If present it specifies the new line number. The new line number is limited to the existing text. char% sets the position in this line (and is limited to the line).

Returns The function returns the new position in the file.

See also:MoveBy(), Selection$()

NextTime()

This finds the next item on a channel after a time. If a marker filter is in use, only data that is included in the filter is visible. It is an error to use this function in a result view.

```
Func NextTime(chan%, time{,&val|code%[]{,data[]|data%[]|&data$}});
```

chan%   The channel number in the view to use.

time   The time to start the search after. Items at the time are ignored. To ensure that items at time 0 are found, set the start time of the search to a negative time.

val   Optional: for waveform channels it returns the waveform value. For event level channels, it returns 0 if the transition is low to high, 1 if the transition is high to low. If there is no event, it returns the level at time; 0 for low, 1 for high.

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.

data   Gets data from RealMark and WaveMark channels. If there is insufficient data, unused entries are unchanged. Integer arrays are for WaveMark channels and return the 16-bit data that holds the waveform. If WaveMark data has multiple traces, use a two dimensional array. The trace number is the second index.

data$ A string returned holding the text from a TextMark channel.

Returns The time of the next item, -1 if there are no more items or a negative error code.

See also:ChanData(), LastTime(), MaxTime()

Optimise()

This optimises y axes over an x axis range in time, result or XY views in the same way as the Y Range dialog optimise button. You can optimise a hidden channel. In an XY view, all channels share the same y range, so optimising affects all channels. You can also use this on a spike shape window (with no arguments) to optimise the display.

```
Proc Optimise({cSpc, start{, finish}}});
```

cSpc   A channel specifier for the source channels or -1 for all, -2 for visible and -3 for selected channels. If omitted, -1 is used, for all channels.

start   The start of the region to optimise. This is in x axis units for a time or XY view and in bins for a result view. 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:ChanOffset(), ChanScale(), YRange(), YLow(), YHigh()

OutputReset()

This command is equivalent to the Output Reset dialogs. It allows you to specify DAC and digital output levels to be set before and after sampling. There are two command versions: the first sets values and the second returns them. The array arguments can be any length without error; extra elements are ignored, missing elements have no effect.

```
func OutputReset(flags%, dacs%[], dacv[],dig%[]{,rampt{,n1401%}});
func OutputReset(&dacs%[], &dacv[], &dig%[]{, &rampt{, n1401%}})
```

flags%   When to apply. Sum of: 1 = at load/program start, 2 = before sampling, 4 = after

dacs%   An array of up to 8 elements. Element n corresponds to DAC n. Set 1 to apply, 0 to not apply the corresponding dacv value.

dacv   An array of up to 8 DAC values in Volts to apply if the corresponding element of dacs%[] is not zero.

dig%   An array of up to 16 elements, element n corresponding to digital output bit n. Set values as: 1=high, 0=low, -1 no change. Elements 0-7 correspond with the output sequencer DIGLOW command, elements 8-15 correspond with DIGOUT.

rampt   Ramp time in seconds, default 0. This relates to a currently unimplemented feature. It will set how long to take ramping DAC outputs to their final values.

n1401%   Currently, set to 0 or omit to set/get values in the sampling configuration and set 1024 for the preferences. We have plans for sampling with multiple 1401s. When this is enabled we will add the 1401 number to n1401%. If n1401% is omitted, this value is taken as 0.

Returns Both function versions return the flag% value at the time of the call.

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.

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:ChanColour(), Colour(), PaletteSet(), XYColour()

PaletteSet()

This sets the colour of one of the 40 palette colours. Colours 0 to 6 form a grey scale and cannot be changed. Colours are specified using the RGB (Red, Green, Blue) colour model. For example, bright blue is (0%, 0%, 100%). Bright yellow is (100%, 100%, 0%). Black is (0%, 0%, 0%) and white is (100%, 100%, 100%).

```
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, sets the nearest solid colour (all pixels have the same hue in a solid colour). Systems that don't need to do this ignore solid%.

See also:ChanColour(), Colour(), PaletteGet(), XYColour()

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[][] A `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 optional 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.

You can find an overview of PCA in the *Clustering spikes* chapter of the Spike2 manual. In the terms of that overview, `x[][]` corresponds with the **X** matrix on input and the **U** matrix on output, `w[]` is the diagonal of **W** and `v[][]` is the matrix **V**.

PlayOffline()

This command plays an area of the current time view through the 1401 DACs or through the sound card in the computer. If you close the view during sampling, output will cease. You cannot use the 1401 for sampling and `PlayOffline()` simultaneously. You cannot use the sound card if it is already in use. If you choose the sound card, the default wave out device is used.

Unless the waveform to replay is small (total points less than 32000), it is copied to the output device in chunks, as required. This operation happens in the background, but you must make time for it by allowing Spike2 to idle, or by checking the play position.

There are two versions of the command; the first starts output and the second reports the play position and allows you to stop the output early.

```
func PlayOffline(cSpc,dacs%,sTime,eTime{,rep%{,scale{,flag%}}});
func PlayOffline({what%{, &rep%}});
```

cSpc A channel specification for up to 4 channels to be played. These can be any mix of waveform, WaveMark or RealWave channels. The channels can have different sample rates; the rate of the first channel is taken as the rate of all channels and the data for the other channels is matched to the first by linear interpolation. Gaps in the data are output as zeros.

dacs% Either an integer array of 1401 DAC channel numbers (the first DAC is 0), or an integer, being the number of the first DAC channel to use. If a single integer is used, and more than 1 channel is in use, the second, third and fourth channels use DACs `(dacs%+1) mod 4`, `(dacs%+2) mod 4`, `(dacs%+3) mod 4`.

If the DAC number for the first channel is negative, output is to the system wave out device (usually a sound card). The maximum number of channels allowed depends on your system, but will usually be at least 2.

sTime The start time in the current time view. This, together with `eTime` defines the data region to play.

eTime The end time of the region in the current time view to output.

rep% In the setup call, this sets the number of times to play the output. If omitted, the value 1 is used. To play for as many repeats as possible use the value 0. The

total number of points to play is limited to 2 to the power 32 (about 4 billion). In the status call, this returns the number of repeats completed.

scale   You can scale the replay rate in the range 0.25 to 4.0 (quarter speed to four times the channel rate). If you omit this argument, 1 is used, for output at the rate of the first channel.

flag%   The sum of: 1 to set cursor 0 at the replay position, 2 to rerun the file linked to the waveform output, 4 to broadcast cursor 0 changes and 8 (needs 4 set) to trigger active cursor searches on cursor 0 changes. The cursor is displayed if it is hidden. When the play finishes, the cursor is hidden if it was originally hidden.

what%   Set this to −1 to stop output. Omit or set to 0 to report the position only. Both forms of the command return the position at the time of the call or −1 if there was no play in progress.

Returns The setup call returns 0 if all was OK, else −1 if there is a problem with the number of channels or DAC channels, -2 for an internal setup problem, -3 if the output rate is too fast, -4 if the rate is too fast to replay in chunks, -5 if memory was exhausted, -6 for a problem with the sound card, -7 for a stupid points count. The status call returns the replay position, in seconds, or −1 if there is no replay in progress, or it has finished.

See also:`DlgShow()`, `Interact()`, `PlayWaveAdd()`, `ReRun()`, `Toolbar()`, `Yield()`

PlayWaveAdd()

This command adds a new area to the on-line play wave list. When you create a data file, Spike2 reserves 1401 memory and transfers any stored waveform to it. The area is set to play once and is not linked to any other area. The area replay speed factor is set to 1.0 and the wave is set to non-triggered. You must use this command before you use `FileNew()` to create the sampling window.

There are command variants to add a wave from the current time view or a Spike2 data file, to add a wave from a data array, and to reserve space without setting any data:

```
func PlayWaveAdd(key$, lb$, dac%, sT, eT, wch%{, mem% {,path$}});
func PlayWaveAdd(key$, lb$, dac%, rate, data{%}[]{[]});
func PlayWaveAdd(key$, lb$, dac%, rate, size%);
```

key$ The first character of `key$` identifies this wave and triggers the wave playing in `SampleKey()`. It is an error to use `Chr$(0)` or an empty string as a key.

lb$ The label for the play wave control bar button that will play this wave, and record the character code as a keyboard marker. Labels can be up to 7 characters long. If you include `&` as a character, it will not appear on the button, but the next character will be underlined and can be used as a keyboard short-cut.

dac%   Either a single DAC channel number or an array of DAC channel numbers. These are the outputs that will be used to play the data. The channel numbers must be in the range 0 to 3, and if more than one channel is specified, the channel numbers must be different.

sT,eT   The start and end times of the data in either the current time view, or in the file identified by the `path$` variable to be used as a source of output data.

wch%   Either a single channel, or an array of channels to use as a data source for playing. There must be one source channel for each output channel set by the `dac%` variable. The channels can be either waveform or WaveMark data. The sample rate is taken from the sampling rate of the first channel in the list. If subsequent channels in the list have different rates, data is interpolated.

mem%   If present, and non-zero, the data is converted to a memory image and becomes independent of the data file. Otherwise, Spike2 stores the file name and extracts the data as required for sampling (so `path$` must be supplied or the current time view file must have been saved to disk).

path$ This optional argument sets the name of the file to extract data from. If absent, the current time view data file is used as the data source. The named file must exist on disk and hold suitable data channels.

rate   When the data does not come from a file, this value sets the sample rate for each channel in Hz. Spike2 will get as close to this rate as it can.

data   This is either an integer or a real array. If there is more than one DAC channel to play, the array must have the same number of rows as there are DACs, that is if there are three DACs, the array must be equivalent to var data[n][3]; where n is the number of data points. If this is an integer array, the bottom 16 bits of each element is played through the DACs. If the data is in a real array, we assume that the full range of the DACs is ±5 Volts and that the data is the required output value in Volts.

size%   This is the number of data points per channel to reserve for this area. The data values are not specified and you must use PlayWaveCopy() to transfer data to the 1401 for playing after sampling has started.

Returns The memory bytes in the 1401 used to hold this data, or a negative error code.

This command does not transfer the data to the 1401; that happens when the start sampling command is given. Spike2 always keeps a minimum memory area for data sampling, so there is a limit to size of the waveforms that can be copied to the 1401. This size limit is not known until sampling starts. There is also a limit on the size of a waveform that can be stored in the list, which is 32,000,000 bytes.

There are PlayWave...() commands to link areas together, set the area repeats and change the area speed. SampleClear() removes all stored waveforms.

To start a waveform playing from the script you can use the SampleKey() function with the same key as set for the area or the output sequencer WAVEGO instruction.

See also:PlayWaveDelete(), PlayWaveLabel$(), PlayWaveLink$()

PlayWaveChans()

This function lets you read back or set the DAC channels assigned to a particular play wave area. You cannot change the DAC assignments after sampling has started.

```
func PlayWaveChans(key${, ch%[] {, set%}});
```

key$ The first character of the string identifies the play wave area.

ch%   An optional integer array used to collect or set the DAC channel numbers. The array size must match the number of channels.

set%   If omitted or 0, the ch% array is filled in with the DAC channels used. If non-zero, the DAC channels are changed to the list defined by the ch% argument.

Returns The command returns the number of DACs in the area or a negative error code.

See also:PlayWaveAdd(), PlayWaveInfo$(), PlayWaveLabel$(),
          PlayWaveLink$(), PlayWaveRate()

PlayWaveCopy()

This command is used to update a play wave data area in the 1401 memory. This can be done at any time that SampleStatus() returns 0 or 2, even while a wave is playing.

```
func PlayWaveCopy(key$, data{%}[]{[]}{, offs%});
```

key$ The first character of the string identifies the area to be updated.

data   This is either an integer or a real array. If there is more than one DAC channel to play, the array must have the same number of columns as there are DACs, that is

if there are three DACs, the array must be equivalent to `var data[n][3];` where `n` is the number of data points. If this is an integer array, the bottom 16 bits of each element is played through the DACs. If the data is in a real array, we assume that the full range of the DACs is ±5 Volts and that the data is the required output value in Volts. The data in the array is copied to 1401 memory. It is an error for the array size to be larger than the memory area in the 1401.

offs%   The destination offset within the data area, in data points per channel. If the size of the data is such that the copy operation would extend beyond the end of the target area, the extra data is copied to the start of the area. The first offset is 0. Use `PlayWaveStatus$()` to find the next offset to be written to the DACs.

Returns   The function returns 0 if all went well or if you call when there is no sampling or sampling is stopping. It returns a negative error code if there is a problem or you have requested sampling to start and it hasn't yet started.

See also:`PlayWaveCycles()`, `PlayWaveLink$()`, `PlayWaveSpeed()`, `PlayWaveStatus$()`, `PlayWaveStop()`, `SampleStatus()`

PlayWaveCycles()

This function gets and sets the number of times to play a waveform area associated with a particular key. If this is used on-line, it will also change the number of repeats for the next play of the waveform.

```
func PlayWaveCycles(key$, {new%});
```

key$ The first character of the string identifies the play wave area.

new%   If present, this sets the number of cycles to play. The value 0 sets a very large number of cycles.

Returns   The number of cycles set at the time of the call.

See also:`PlayWaveAdd()`,`PlayWaveLabel$()`,`PlayWaveLink$()`, `PlayWaveStop()`

PlayWaveDelete()

This function deletes one or more play wave areas from the sampling configuration. If you do this while sampling, it will not change the waves loaded to the 1401.

```
func PlayWaveDelete({keys$});
```

keys$ If omitted, all play wave areas are deleted. If present, all areas with a key that is in this string are deleted. Case is significant; `"abc"` and `"ABC"` are not the same.

Returns   The number of areas deleted or a negative error code.

See also:`PlayWaveAdd()`, `SampleClear()`

PlayWaveEnable()

This function reports on the enabled state of a play wave area and optionally enables and disables it. Enabled areas are setup in the 1401 when sampling starts. Changes made with this function once sampling has started will not change the waves loaded to the 1401.

```
func PlayWaveEnable(keys$,{set%});
```

key$ The first character of the string identifies the play wave area.

set%   If present and zero, the area is disabled. If non-zero, the area is enabled.

Returns   The enabled state (1=enabled, 0=disabled) of the area at the time of the call, or a negative error code.

See also:`PlayWaveAdd()`, `PlayWaveInfo$()`, `PlayWaveStatus$()`

PlayWaveInfo$()

This function returns the list of keys associated with play wave areas, or gets the type of a particular area and the name of any associated data file.

```
func PlayWaveInfo$({key$, &size%, &type%});
```

key$ If omitted, the function returns the list of keys associated with the play wave areas. If present, information about a specific area is returned in the remaining arguments and the function returns any file name associated with the area.

size%   If present, this is returned as the size of the data area in points per channel. This is not changed by PlayWavePoints() which sets the size actually used.

type%   This returns the type of the area as: 0= unused, 1 = data is taken from a data file, 2 = area has a memory image of data (and may also have an associated data file), 3 = area is reserved by the script but there is no associated data.

Returns If there is no key$ value, then a string is returned holding one key character for each area. If there is a key, then the function returns the name of any data file associated with the area, in which case type% will be returned as 1 or 2.

See also:PlayWaveChans(), PlayWaveCycles(), PlayWaveLabel$(),
        PlayWaveLink$(), PlayWaveStatus$()

PlayWaveKey2$()

While sampling, you can define a second, sub-key, that can be used to identify an area for playing with SampleKey() or interactively. This can be useful when you reuse an area to have different contents or change an area size and you want to use a different key code to indicate this. It is an error to use this command when not sampling. The original key code for the area still works if you set a second key.

```
func PlayWaveKey2$(key${, key2$});
```

key$ The first character of the string identifies the play wave area.

key2$ If present, the first character of this string sets the sub key. This can be an empty string to clear the sub key code. If key2$ matches an area code, or the sub key code for any area, no change is made.

Returns The sub-key code that was active at the time of the call or an empty string if there was no sub-key.

See also:PlayWaveAdd(), PlayWavePoints(), SampleKey()

PlayWaveLabel$()

This function returns and/or changes the label associated with each play area. The label can be up to 7 characters long and is used to label the buttons that appear on the Play waveform control bar. If you include & as a character, it does not appear in the label and the next character is underlined and can be used as a short cut to the button when the control bar is the current window.

```
func PlayWaveLabel$(key$, {new$});
```

key$ The first character of the string identifies the play wave area.

new$ If this is present, the label for the area is changed to the string. If the string is more than 7 characters long, only the first 7 are used.

Returns The label at the time of the function call.

See also:PlayWaveAdd()

PlayWaveLink$()

This links play wave areas together. Linked areas must have the same number of output channels and the same output channel list. The sample rate used is the sample rate for the

area that is played first. You can change links during replay if `SampleStatus()` is 0 or 2. Both the area to link from and the area to link to must exist at the time of the call.

```
func PlayWaveLink$(key${, to$});
```

key$ The first character of the string identifies the play wave area.

to$ If present, the first character of this string sets the area to link to. Use `Chr$(0)` to cancel the link from the area set by `key$`.

Returns The key character of the area that was linked at the time of the call or an empty string if no area was linked or there was an error.

See also: `PlayWaveAdd()`, `PlayWaveStatus$()`, `SampleStatus()`

PlayWavePoints()

This gets and sets the size of a waveform area associated with a particular key while sampling; it is an error to use this when not sampling. You can set sizes up to the original size of the area. If you change an area while it is playing, output continues from the start of the area. This command does not affect the cycle count or any other feature of the area.

```
func PlayWavePoints(key${, new%});
```

key$ The first character of the string identifies the play wave area.

new%   If present, this sets the new size in points per channel. If `new%` is zero or greater that the area size, the original size is restored. Negative values generate an error.

Returns The size of the area in points per channel at the time of the call.

See also: `PlayAreaAdd()`, `PlayWaveCopy()`, `PlayWaveKey2$()`

PlayWaveRate()

This function gets or sets the base play rate for a play wave area. This is the standard play rate that can be changed by `PlayWaveSpeed()`. Changes to the rate made after sampling starts have no effect on the output; use `PlayWaveSpeed()` for on-line changes.

```
Func PlayWaveRate(key$, {new});
```

key$ The first character of the string identifies the play wave area.

new   If present, this is the new play rate for the area, in samples per second. You can set any value you like (not 0 or negative) and Spike2 will get as close as it can with the available hardware.

Returns The rate for the channel at the time of the function call.

See also: `PlayWaveAdd()`, `PlayWaveLink$()`, `PlayWaveSpeed()`, `SampleKey()`

PlayWaveSpeed()

You can alter the sample rate for a play wave area by a factor of 0.25 to 4.0 with this command. Spike2 may not be able to play at the rate you request; it will set the closest rate it can. The `Hz` argument returns the achieved rate. On-line changes are allowed.

```
func PlayWaveSpeed(key${, new{, wait%{, Hz}}});
```

key$ The first character of the string identifies the play wave area. If this area is playing, or an area that this area links to, the rate will change during playing.

new   If present, this is the new speed factor for the area, in the range 0.25 to 4.0. Spike2 gets as close to this speed factor as it can with the available hardware.

wait%   If present and non-zero, any on-line speed change is postponed until the end of the current cycle and will happen within a few milliseconds of the cycle end.

Hz   If present and a sampling document is open, it returns the real replay rate in Hz.

Returns The speed factor for the area at the time of the function call or 0 if there is no area defined by the key.

See also: PlayWaveAdd(), PlayWaveLink$(), PlayWaveRate(), SampleKey()

PlayWaveStatus$()

This function returns information about waveform output during sampling.

```
func PlayWaveStatus$({&pos%{, &cyc%}}});
```

pos%   If present, this returns the next position, in terms of the number of points per channel in the area, to write to the DACs. The first position is 0. The DACs hold one data point ready for output on the next clock tick, so pos% is the next but one index to output. If you play an area with a triggered start but do not trigger it, pos% is 1, not 0. This is because the DACs are already holding index 0, ready to output when the trigger arrives.

cyc%   If present, this integer variable is returned holding the number of cycles left to play (including the current cycle).

Returns The key code of the area that is playing or waiting for a trigger, or an empty string if no area is playing or sampling is not active.

See also: PlayWaveAdd(), PlayWaveCycles(), PlayWaveLink$(), PlayWaveRate(), PlayWaveSpeed(), PlayWaveStop(), SampleKey()

PlayWaveStop()

This function requests that the currently playing wave is stopped, either immediately, or when the current cycle finishes.

```
func PlayWaveStop(cEnd%);
```

cEnd%   If present and non-zero, the current cycle for the playing area will be the last cycle for that area, otherwise output will stop immediately.

Returns 1 for OK, 0 if not playing or a negative error code.

See also: PlayWaveAdd(), PlayWaveCycles(), PlayWaveStatus$(), SampleKey()

PlayWaveTrigger()

This function reports and optionally changes the trigger state of a play wave area. If an area is triggered, a play request prepares the area but output does not start until a trigger signal is received by the 1401. This trigger is the E3 front panel input for the 1401*plus* and is the Trigger input for the Micro1401 and Power1401 unless it is routed to the rear panel by the Edit Preferences menu.

```
func PlayWaveTrigger(keys$,{set%});
```

key$ The first character of the string identifies the play wave area.

set%   If present this sets the triggered state. 0 = not triggered and non-zero = triggered.

Returns The trigger state (1=triggered, 0=not triggered) of the area matching the first character in keys$ at the time of the call, or a negative error code.

See also: PlayWaveAdd(), PlayWaveEnable(), PlayWaveStatus$()

Pow()

This function raises x to the power of y. If the calculation underflows, the result is 0.

```
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.

See also:Abs(), ATan(), Cos(), Exp(), Ln(), Log(), Sin(), Sqrt(), Tan()

Print()

This command prints to the current view at the text caret. If the first argument is a string (not an array), it is used as 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, Spike2 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 following arguments. The string contains two types of characters: ordinary text that is copied to the output unchanged and format specifiers that convert the following arguments to text. Format specifiers start with % and end with one of the letters d, x, c, s, f, e or g in upper or lower case. For a % in the output, use %% 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 on the left to the field width 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 present, it is a number that sets the minimum output field width. 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 and 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.ddddddde±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 Spike2 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:`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 with formatting information. See `Print()` for a description.

arg1,2   The data to format 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:`Asc(), Chr$(), DelStr$(), LCase$(), Left$(), Len(), Mid$(), Print(),`
`PrintLog(), Right$(), Str$(), UCase$(), Val()`

PrintLog()

This command prints to the log window. The syntax is identical to `Print()`.

```
Func PrintLog(form$|arg0 {,arg1 {,arg2...}});
```

form$ An optional string with formatting information. See `Print()` for a description.

arg1,2   The data to print.

Returns   0 or a negative error code. Fields that cannot be printed are filled with asterisks.

See also: `Print()`, `Print$()`, `Message()`

Process()

Processes the current view. For result and XY views, the time view to process data from must be open. The times for start and end of processing are times in the time view. Use `View(-1).Cursor(1)` to refer to time view times from result and XY views.

```
Func Process(sTime, eTime{, clear%{, opt%{, dest%{,
                                        gate%, len, pre{, mCd%}}}});
```

sTime   The time to start processing from, in seconds. Negative times are treated as zero. Times greater than `Maxtime()` cause no processing. In triggered modes with no trigger channel, this sets the trigger time and `eTime` is ignored. In Phase histograms with no Cycle channel, this sets the start of a cycle.

eTime   The end time for processing. In Phase histogram mode with no Cycle channel, this sets the end time of a single cycle.

clear%   If present, and non-zero, the result view bins are cleared before the results of the analysis are added to the result view and `Sweeps()` result is reset.

opt%   If present and non-zero, the result view is optimised after processing the data.

dest%   Used when processing to a channel from `MeasureToChan()`. If not used it should be set to 0. This identifies the destination channel for processing. If omitted or 0, all suitable channels are processed. It is not an error if this doesn't match a destination channel, but no processing happens.

gate%   If present, this is the channel number of an event or marker-based channel in the associated time view to use as a gate. Both `len` and `pre` must be present.

len   The time to process for each gate event in the time `sTime` to `eTime`.

pre   How far before each gate event to start processing.

mCd%   If present and `gate%` is a marker or derived type, it holds the marker code to use as the gate. Set this to -1 or omit it to use the current channel marker filter.

Returns   This returns the number of items processed. This is the number of intervals considered for an INTH (even if they didn't fall in the histogram), the number of sweeps for sweep-based analysis, the number of data blocks for `SetPower()`. In the case of an error, a negative error code is returned.

See also: `MeasureToChan()`, `MeasureToXY()`, `ProcessAll()`, `SetAverage()`, `SetEvtCrl()`, `SetINTH()`, `SetPhase()`, `SetPSTH()`, `SetPower()`, `SetResult()`, `SetWaveCrl()`, `Sweeps()`

ProcessAll()

This function runs all processes that use the current time view as a data source. It is as if a `Process()` command were used for each target view using the current processing settings. These are the settings you would see if you opened the Process Settings dialog for the target. If you have already used a `Process()` command on a target, this sets the current processing settings.

```
Func ProcessAll(sTime, eTime);
```

sTime   The time to start processing from, in seconds. Negative times are treated as zero. Times greater than `Maxtime()` cause no processing. In triggered modes with no trigger channel, this sets the trigger time and `eTime` is ignored. In Phase histograms with no Cycle channel, this sets the start of a cycle.

eTime   The end time for processing. In Phase histogram mode with no Cycle channel, this sets the end time of a single cycle.

Returns Zero if no errors or a negative error code.

See also:`MeasureToChan()`, `Process()`, `MeasureToXY()`, `SetAverage()`, `SetEvtCrl()`, `SetINTH()`, `SetPhase()`, `SetPSTH()`, `SetPower()`, `SetResult()`, `SetWaveCrl()`, `Sweeps()`

ProcessAuto()

This is equivalent to the Process Dialog for a New file in Automatic mode. The current view must be the view where the results appear. The processing parameters set by this command are used when this view is given a chance to update.

```
Func ProcessAuto(delay, mode%{, opt%{, last{, leeway{, dest%}}}});
```

delay   The minimum time between updates, in seconds.

mode%   0=accumulate all data. 1=clear the result, process the most recent `last` seconds.

opt%   If present and non-zero, the result view is optimised after each process.

last   The length of time to process in mode 1, in seconds; ignored in mode 0.

leeway How close cursor 0 can be to the file end for XY views and `MeasureToChan()`.

dest% The channel for `MeasureToChan()`, ignored for other process types. If omitted or 0, all suitable channels are processed. It is not an error if this doesn't match a destination channel, but no processing happens.

Returns 0 or a negative error code.

See also:`MeasureToChan()`, `MeasureToXY()`, `ProcessTriggered()`, `Process()`

ProcessTriggered()

This is equivalent to the Process Dialog used with a New file in Gated mode. The current view must be the view where the results appear. The processing parameters set by this command are used when this view is given a chance to update.

```
Func ProcessTriggered(len,pre,gate%{,clr%{,opt%,{mCd%{,dest%}}}});
```

len   The length of data to process around each gate event, in seconds.

pre   The pre-event time, in seconds.

gate%   The channel holding gate events or markers.

clr%   If present, and non-zero, the result view bins are cleared before the results of the analysis are added to the result view and `Sweeps()` result is reset.

opt%   If present and non-zero, the result view is optimised after processing the data.

mCd%   If present and `gate%` is a marker or derived type, it holds the marker code to use as the gate. Set this to -1 or omit it to use the current channel marker filter.

dest%   The channel for `MeasureToChan()`, ignored for other process types. If omitted or 0, all suitable channels are processed. It is not an error if this doesn't match a destination channel, but no processing happens.

Returns 0 or a negative error code.

See also:`MeasureToChan()`, `MeasureToXY()`, `ProcessTriggered()`, `Process()`

Profile()

Spike2 saves information in the `HKEY_CURRENT_USER\Software\CED\Spike2` section of the system registry. The registry is a tree of keys with 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. The `Profile()` command manipulates the keys and values within the `Spike2` section of the registry.

You can also 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 using this program. It is a very powerful tool; 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 Spike2 section of the registry, use `regedt32` to locate the `Spike2` section and delete it. The next time you run Spike2 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 sets the key to use in the `Spike2` registry section. Set `key$` empty to use the `Spike2` key. 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 to 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 is either a string or an integer. If `read` is omitted, `val` is written to the registry. If `read` is present, `val` is returned 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 `name$` is not in the registry, `read` is set to `val`.

`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$` is returned in `read%` or `read$`.

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 Spike2 `HKEY_CURRENT_USER\Software\CED\Spike2` holds the following keys:

BarList This key holds the list of scripts to load into the script bar when Spike starts.

Edit 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 and 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, Spike2 will use the changed information immediately. The values are all integers except the file path, which is a string:

E3 trigger at rear	0=Sample and `PlayWave` trigger on front panel, 1=on rear
Enhanced Metafile	0=Windows metafile, 1=enhanced metafile for clipboard.
Enter debug on error	0=Do not enter debug, 1=enter debug

Event ports at rear	0=Events 0&1 on front panel, 1=on rear panel.
Fast online scroll	0=Normal, 1=faster (inaccurate) scrolling algorithm
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 NoCompress	0=Compress when multiple points per x pixel, 1=no compress.
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 Save Prompt	0=Prompt to save derived views, 1=no prompt.
Save modified scripts	0=Do not save, 1=save
Ten volt 1401	0=5 Volt 1401, 1=10 Volt 1401, 2=same as last 1401.
Use colour	0=Use Black and White, 1=Use Colour (from the View menu).

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 Spike2.

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. The colour palette is also saved here. Set "No MThread Sampling" to 1 to force single threaded sampling.

Tip The *Tip of the Day* dialog uses this key to remember the last tip position.

Version Spike2 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 Help menu About Spike2 dialog displays the current working set sizes. Click Help in this dialog to read more about using these registry values.

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 can be dangerous, as it will terminate a program without giving it the opportunity to save data.

```
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 command prompt. Use `ProgStatus()` to test the program status, `ProgKill()` to terminate it. The program inherits its environment variables from Spike2, see `System$()` for details.

```
Func ProgRun(cmd$ {,code% {,xLow, yLow, xHigh, yHigh}});
```

cmd$ The command string as typed at a command prompt. To run shell command x use "`cmd /c x`" in Windows NT and XP, "`command /c x`" in Windows 9x. To run another copy of Spike2, use "`sonview.exe /M {filename}`".

code%   If present, this sets the initial application window state: 0=Hidden, 1=Normal, 2=Iconised, 3=maximised. Some programs set their own window state 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. `ProgStatus()` releases resources associated with the handle when it detects that the program has terminated.

See also: `FileCopy()`, `FileDelete()`, `ProgKill()`, `ProgStatus()`, `System$()`

ProgStatus()

This function tests if a program started with `ProgRun()` is still running. If it is not, resources associated with the program handle are released.

```
Func ProgStatus(pHdl%);
```

pHdl% The program handle returned by `ProgRun()`.

Returns 1=program is running, 0=terminated, resources released, handle now invalid. A negative error code (-1525) means that the handle is invalid.

See also: `ProgKill()`, `ProgRun()`

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(text$, {,Yes$ {,No$}});
```

text$ This string forms the text in the window. If the string includes a vertical bar, the text before the vertical bar is used as the window title.

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 returns pseudo-random numbers with a uniform density in a set range. The values returned are R * `scl` + `off` where R is in the range 0 up to, but not including, 1. Spike2 initialises the generator with a random seed based on the time. You must set the seed for a repeatable sequence. 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 system time.

arr This 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 Spike2, 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 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 $p(x) = (2\pi)^{-\frac{1}{2}}e^{-x^2/2}$; this has a mean of 0 and a variance of 1. When you start Spike2, 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 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()

RasterAux()

This is now deprecated; use RasterSort() or RasterSymbol() instead. It is here for version 4 compatibility, but will be removed in a future release. This function returns and optionally sets some of the raster auxiliary values for a channel in the current result view.

Values 0 and 1 are used to sort the sweeps and can be selected in addition to time order by the `DrawMode()` command. Unset values read back as zero.

Values 2 to 5 are the times, in seconds relative to the start of the file, of four markers (circle, cross, square and triangle) to display in the sweep. The markers are drawn in the colours set for WaveMark codes 1 to 4. Markers appear if their position relative to the sweep trigger lies within the sweep. Unset values read back as a negative time.

```
Func RasterAux(chan%, sweep%, num% {,new});
```

chan%   The channel in the result view. The channel must have raster data enabled.

sweep% The sweep number in the range 1 to the number of sweeps in the channel.

num%   The auxiliary value to return and optionally change in the range 0 to 5.

new   If present, this changes the auxiliary value for the sweep.

Returns The auxiliary value at the time of the call.

Replace `RasterAux(c%,s%,n%{,new})` with `RasterSort(c%,s%,n%{,new})` if `n%` is 0 or 1, and with `RasterSymbol(c%,s%,n%-2{,new})` if `n%` is 2 to 5.

See also: `DrawMode()`, `RasterGet()`, `RasterSet()`, `RasterSort()`, `RasterSymbol()`, `SetEvtCrl()`, `SetPSTH()`

RasterGet()

This returns a sweep of raster data for a result view channel for which raster data has been enabled. Using this when the current view is not a result view causes a fatal error.

```
Func RasterGet(chan%{,sweep%{,&sT{,data{%}[]{,&tick{,&eT}}}}});
```

chan%   The channel number in the result view. If this is the only argument, the return value is the number of sweeps in the channel.

sweep% The sweep number in the result view in the range 1 to the number of sweeps. If this argument is present, the return value is the number of times in the sweep.

sT   This is returned holding the time in seconds of the start of the sweep in the original data. This is not the sweep trigger time (the time that corresponds to the x axis 0 in the result view). The trigger time is `sT-BinToX(0)`.

data   This optional array is filled with sweep event times. A real array returns times in seconds; an integer array returns times in units of the microseconds per time used when the view was created. The number of times copied into the array is the lesser of the number of times in the sweep and the length of the array.

tick   This optional real value is the number of seconds per time units used when times are returned as an integer array. That is, if you multiply each integer time returned in `data%[]`, you would get the time in seconds. If you created the result view using `SetResult()`, it is the tick value you passed in; otherwise, it is the `BinSize()` value from the associated time view.

eT   This optional real value is returned holding the end time of the sweep, in seconds. This is usually only of interest for data created with `SetPhase()`.

Returns The count of sweeps in the channel, times in the sweep or a negative error code.

See also: `RasterSet()`, `RasterSort()`, `SetEvtCrl()`, `SetPhase()`, `SetPSTH()`

RasterSet()

This function sets the raster data for a sweep for a channel of a result view. You can replace the data for an existing sweep, or add a new sweep. It is a fatal script error to call this function when the current view is not a result view.

```
Func RasterSet(chan%, sweep%, sT {,data[]|data%[] {,eT}});
```

chan%   The channel number in the result view.

sweep%   This is either in the range 1 to Sweeps() to replace the existing raster data for a sweep, or it can be 0 to add a new sweep. If you add a new sweep, *the new sweep is added to all channels*. The remaining channels have a sweep added that has the same start time, and no data. If your result view has multiple channels, only use a sweep% value of 0 for one of the channels.

sT   This sets the time of the start of the sweep, in seconds. This is not the trigger time of the sweep (the time that corresponds to the x axis 0 in the result view). The trigger time is sT-BinToX(0).

data   This is a real or an integer array holding the events times for the sweep. The size of the array sets the number of items. If this is a real array, the times are in seconds. If this is an integer array, the times are in the underlying tick units (see RasterGet() and SetResult() for details). Times outside the time range start to start+MaxTime()*BinSize() are ignored.

eT   This optional value is the end time of the sweep. If you supply this, all times in the data array are mapped into the time period sT to eT, regardless of the x axis scaling set for the result view. This argument is usually only used when working with or emulating a Phase histogram.

Returns The sweep number that received the data or a negative error code.

See also:RasterGet(), RasterSort(), SetEvtCrl(), SetPhase(), SetPSTH()

RasterSort()

Each raster sweep has 4 values that can be used to sort displayed rasters; this function returns and optionally sets these values for the current result view. The DrawMode() command selects between time order and one of these values.

```
Func RasterSort(chan%, sweep%, num% {,new});
```

chan%   The channel in the result view. The channel must have raster data enabled.

sweep%   The sweep number in the range 1 to the number of sweeps in the channel.

num%   The sort value to return and optionally change in the range 1 to 4.

new   If present, this changes sort value num% for the sweep.

Returns The sort value at the time of the call. Unset values read back as zero.

See also:DrawMode(), RasterGet(), RasterSet(), RasterSymbol(), SetEvtCrl(), SetPSTH()

RasterSymbol()

Each raster sweep has 8 values that can be displayed as symbols; this function returns and optionally sets these values for the current result view. The values are times, in seconds relative to the start of the file. The 8 markers are: circle, cross, square, up triangle, plus, diamond, down triangle, filled square. The markers are drawn in the colours set for WaveMark codes 1 to 8. Markers appear if their position relative to the sweep trigger lies within the sweep. Unset values read back as a negative time.

```
Func RasterSymbol(chan%, sweep%, num% {,new});
```

chan%   The channel in the result view. The channel must have raster data enabled.

sweep%   The sweep number in the range 1 to the number of sweeps in the channel.

num%   The symbol number in the range 1 to 8 to get or set.

new   If present, this sets the symbol time for the sweep. Set -1 to cancel the symbol.

Returns The symbol time at the time of the call. Unset symbols have a negative time.

See also:DrawMode(), RasterGet(), RasterSet(), SetEvtCrl(), SetPSTH()

Read()

This function reads the next line from the current text view or external text file and converts the text into variables. The read starts at the beginning of the line containing 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 to convert the string data. In a successful call, each variable matches a field in the string, and the value of the variable changes 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 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. An array of length n is treated as n individual values.

The source string is expected to hold data values as real numbers, integer numbers (decimal or hexadecimal introduced by 0x) and strings. Strings can be delimited by quote marks, for example "This is a string", or they can be just text. If a string is not delimited, it is deemed to run to the end of the source string, so no other items can follow it. You can use ReadSetup() to change the characters that delimit a string and also to define hard separator characters within non-delimited strings. String delimiters are not returned as part of the string.

Normally, 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 hard separators (with or without intervening soft separators), imply a blank field. You can use ReadSetup() to redefine the soft and hard separators. When reading a field, the following rules are followed:

1. Soft separator (space) characters are skipped over
2. If the field is a string and the next character is a delimiter, it is skipped.
3. Characters that are legal for the destination variable are extracted until a non-legal character or a separator or a required string delimiter 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.
4. Characters are skipped until a separator character is found or end of data. If a soft separator is found, it and any further soft separators are skipped. If the next character is a hard separator, it is also skipped.
5. If there are no more variables or no more data, the process stops, else 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;                              This is equivalent to the result
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:EditCopy(), FileOpen(), ReadSetup(), ReadStr(), Selection$()

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()`, `ReadSetup()`, `Val()`

ReRun()

This function controls the rerun of the current time view and is equivalent to the View menu ReRun command. You cannot use this on a file that is being sampled.

```
Func ReRun({run%{, sTime{, eTime{, scale}}}});
```

run%   Set 1=start rerun, 0=stop rerun or omit for no change. Negative values return: -1=sTime, -2=eTime, -3=scale, -4=the rerun time or -1 if not rerunning.

sTime   Sets the rerun start time. If omitted, 0 is used. This is ignored unless $run\% > 0$.

eTime   Sets the rerun end time. If omitted `MaxTime()` is used. Ignored unless $run\% > 0$.

scale   Sets the rerun time scale. If omitted, 1.0 is used. A value of 2 reruns twice as fast. Values from 0.01 to 100.0 are allowed. This is ignored unless $run\% > 0$.

Returns If `run%` is positive, omitted or less than -4, the command returns the state at the time of the call: 0=not rerunning, 1=rerunning, 2=rerunning linked to a play offline waveform output, -1= rerunning is not allowed. Negative values of `run%` return the rerun settings as described for `run%`.

See also:`PlayOffline()`

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:`Asc()`, `Chr$()`, `DelStr$()`, `InStr()`, `LCase$()`, `Left$()`, `Len()`, `Mid$()`, `Print$()`, `ReadStr()`, `Str$()`, `Trim()`, `UCase$()`, `Val()`

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:`Frac()`, `Max()`, `Min()`, `Trunc()`

SampleAbort()	This cancels sampling a file or file sequence and closes any associated data, result and cursor views without saving them. If a result view derived from the data has been saved, the saved file remains. It is equivalent to the Abort button in the sampling control panel.

```
Func SampleAbort();
```

Returns 0 if sampling was aborted, or a negative error code.

See also:`SampleKey()`, `SampleReset()`, `SampleStart()`, `SampleStop()`, `SampleStatus()`, `SampleText()`, `SampleWrite()`

SampleAutoComment()	This gets or sets file auto-commenting state as set in the sampling configuration dialog.

```
Func SampleAutoComment({yes%});
```

yes%   If present a non-zero value turns on automatic prompting for file comments when sampling ends and zero turns it off. If absent, no change is made.

Returns the automatic commenting flag at the time of the function call, 0=off, 1= on.

See also:`FileComment$()`, `SampleAutoFile()`, `SampleAutoName$()`

SampleAutoCommit()	This gets or sets the automatic file commit period as set in the sampling configuration dialog. There is an overhead associated with the commit operation; do not set needlessly short time periods. A period of 0 means no file commit.

```
Func SampleAutoCommit({every%});
```

every% If present this sets the automatic file commit interval in seconds. The value is limited to the range 0 to 1800 seconds (a maximum of 30 minutes).

Returns the automatic commit period in seconds at the time of the function call.

See also:`FileSave()`, `SampleAutoFile()`, `SampleWrite()`

SampleAutoFile()

This gets or sets the automatic filing state as set in the sampling configuration dialog.

```
Func SampleAutoFile({yes%});
```

yes%   If present, a non-zero value turns on automatic data filing when sampling ends and zero turns it off. If absent, no change is made.

Returns the automatic filing state at the time of the function call, 0=off, 1= on.

See also:FileSave(), FileSaveAs(), FilePathSet(), SampleAutoName$(), SampleAutoComment(), SampleRepeats()

SampleAutoName$()

This gets or sets the template for file auto-naming as in the sampling configuration dialog. The path is set by FilePathSet(). The name must be set before you open the file for sampling for the automatic name to be used.

```
Func SampleAutoName$({name$});
```

name$ If present, this is the template for file auto-naming. An empty string turns off auto-naming. The maximum name length is 23 characters. 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:FilePathSet(), SampleAutoFile(), SampleAutoComment()

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 .s2c file extension, followed by a vertical bar then, a comment to display when the mouse pointer is over the button. Call the command with no arguments to return the number of toolbar buttons.

```
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:\\Spike3\\Fast.s2c|Fast 4 channel sampling");
SampleBar("Faster|C:\\Spike3\\FastXX.s2c|Very fast sampling");
```

See also:App()

SampleBigFile()

This function returns and optionally sets the big file flag in the sampling configuration.

```
Func SampleBigFile({big%});
```

big%   If present, 1 sets the big file flag, 0 clears it. If omitted, it is unchanged.

Returns The big file flag from the sampling configuration at the time of the call.

See also:FileNew(), FileSaveAs(), SampleClear()

SampleCalibrate()

This sets or gets the waveform or WaveMark channel calibration. It changes or reads the units, scale and offset fields only. See the Sampling Configuration dialog for a description of these fields. When used to read back information, the unit$, scale and offset arguments must be variables:

```
Func SampleCalibrate(chan%, units$, scale, offset{, get%});
Func SampleCalibrate(chan%, &units$, &scale, &offset, 1);
```

chan%   The channel number of a waveform or WaveMark channel.

units$ The new channel units (up to 5 characters) or a string variable to read the units.

scale   The new scale factor for the channel or a real variable to read back the scale.

offset The new offset for the channel or a real variable to read back the offset.

get%   Omit or set to 0 to set the calibration, set to 1 to read it back.

Returns 0 if all was well, or a negative error code if the channel is the wrong type.

See also:SampleTimePerAdc(), SampleTitle$(), SampleUsPerTime(),
        SampleWaveform(), SampleWaveMark()

SampleChanInfo()

This function returns channel information that is common to most channels from the current sampling configuration. It also reports the fixed channel numbers that are associated with the special channels for keyboard, text and digital markers.

```
Func SampleChanInfo(chan%{, item%});
```

chan%   A channel number in the sampling configuration in the range 1-100. The value 0 is reserved. Negative values return the channel numbers of special channels:
        -1 the Keyboard channel
        -2 the TextMark channel
        -3 the Digital marker channel.

item%   If omitted, this takes the value 0, otherwise it determines the information to return. When used with chan% greater than 0 the information returned is:
        0 Channel type as returned by ChanKind().
        1 Port number used by the channel.
        2 Expected event rate (same as waveform rate for waveform channel).
        3 Ideal waveform rate (same as event rate for event channel). Items 2 and 3 differ for WaveMark channels.
        4 Actual waveform rate for waveform and WaveMark channels, 0 for others.

        When used with negative value of chan%:
        0 Return the special channel number even if it is not used.
        1 Return the special channel number if it is used, or 0 if it is not used

Returns The information requested or a negative error code.

See also:ChanKind(), SampleCalibrate(), SampleTextMark(),
        SampleWaveMark()

SampleChannels()

This function returns and optionally sets the maximum number of channels to sample. The next time you sample, the data file will have space for this number of channels. Spike2 versions 6 and 7 allow up to 400 channels. Version 5 allows up to 256 (but can read up to 400). Version 4 reads files with up to 100 channels. Version 3 can only read files with 32 channels. The maximum number of channels that you can sample is 100.

```
Func SampleChannels({nChan%});
```

nChan% If present, this sets the number of channels in the next data file created for sampling in the range 32-400. Numbers outside this range are set to the nearer

limit. The maximum channel number for sampling is the smaller of the maximum channels in the file or 100.

Returns The number of channels set at the time of the call.

See also:`FileNew()`, `FileSaveAs()`, `SampleClear()`

SampleClear()

This either turns one channel Off, or resets the sampling configuration to: 32 channels (all Off except the keyboard), sampling mode is Continuous, microseconds per time is 10, time per ADC is 10, sequence files are disconnected, *Stop sampling when...* fields are disabled and the automatic file name template is cleared and all sample rate optimising is disabled (equivalent to `SampleOptimise(0,0,0,2,50)`).

```
Proc SampleClear({chan%});
```

chan% If present, the channel is removed from the sampling configuration (set to Off). If absent, the entire sampling configuration is reset. The option to clear a single channel was added at version 7.00.

See also:`SampleLimitSize()`, `SampleLimitTime()`, `SampleOptimise()`,
        `SampleTimePerAdc()`, `SampleUsPerTime()`

SampleComment$()

This function gets and sets the comment attached to a channel in the Sampling Configuration dialog and the label and comment used by the Sample Bar (also visible in Windows Explorer when inspecting Spike2 configuration files).

```
Func SampleComment$(chan% {,new$});
```

chan% The channel number in the window (1 to 100) or 0 for the Sample Bar comment or -1 for the Sample Bar label.

new$ If present, the new comment or label. If the text is too long, it is truncated.

Returns The original comment or label.

See also:`SampleDigMark()`, `SampleEvent()`, `SampleTextMark()`,
        `SampleTitle$()`, `SampleWaveform()`, `SampleWaveMark()`

SampleConfig$()

This function gets information about the current sampling configuration as text.

```
Func SampleConfig$({get%});
```

get%   What to get, taken as 0 if omitted. Values are:
       0 The name of the file from which the configuration was loaded or an empty string if the name is unknown. Sampling clears the file name as sampling can change the configuration. The file may have a `.CFG` or `.SMR` file extension.
       1 The configuration rendered as text using spaces to align data.
       2 As 1 using commas to separate fields.
       3 As 1 using Tab characters to separate fields.

Returns The information requested by the `get%` argument.

See also:`SampleChanInfo()`, `SampleMode()`

SampleDebounce()

This gets and or sets the debounce time in the range -1 to 1000.0 milliseconds for Event-, Event+ and digital marker channels. Events closer than this are filtered out during sampling. Set -1 to convert simultaneous events to have consecutive clock tick times.

```
Func SampleDebounce(chan%{, ms)
```

chan%   The channel number in the sampling configuration.

ms   If present, the new debounce period in milliseconds. Set 0 for no debouncing.

Returns The previous value, in milliseconds.

See also:SampleDigMark(), SampleEvent()

SampleDigMark()

This adds the digital marker channel to the sampling configuration. The channel title and comment are not changed; set these with SampleTitle$() and SampleComment$().

```
Func SampleDigMark(rate)
```

rate   The expected sustained rate for digital markers on this channel in Hz.

Returns 0 if all went well, or a negative error code.

See also:SampleEvent(), SampleTextMark(), SampleTitle$()

SampleEvent()

This function sets a channel to sample event data. The channel title and comment are not changed; you can set these with SampleTitle$() and SampleComment$().

```
Func SampleEvent(chan%, port%, type%, rate);
```

chan%   The channel number in the file to use for this data (1-100).

port%   The event port number.

type%   The type of the event channel: 0=Events on a falling edge (Event-), 1=Events on a rising edge (Event +), 2=Events on both edges (Level)

rate   The expected maximum sustained event rate on the channel in Hz.

Returns 0 if all went well, or a negative error code.

See also:SampleClear(), SampleDigMark(), SampleTitle$()

SampleHandle()

Returns view handles linked to the current sampling view (which must exist). This can be used to position, show and hide the output sequencer and sampling control panels. App() also returns control panel handles and does not require a sampling view to exist.

```
Func SampleHandle(which%);
```

which% Selects which view handle to return:
    0 Sampling time view   1 Sampling control panel   2 Sequencer control panel
    3 Sample Status bar

Returns The view handle or 0 if the view does not exist or if there is no sampling view.

See also:App(), View(), ViewList(), Window(), WindowVisible()

SampleKey()

Add an event to the keyboard marker channel as if typed, including triggering the output sequencer and arbitrary waveform output. There was no return value before version 5.04.

```
Func SampleKey(key$);
```

key$ The first character of the string is added to the keyboard marker channel.

Returns The time stamp of the added marker in seconds or -1 if no file is sampling.

See also:PlayWaveAdd(), SampleText(), SampleWrite()

SampleKeyMark()

This restores the keyboard marker channel to the sampling configuration. The keyboard marker channel is automatically present in new sampling configurations and this command is only needed in rare cases when a user has imported a sampling configuration from a data file with no keyboard marker channel.

```
Func SampleKeyMark({rate})
```

rate   The expected sustained rate for keyboard markers on this channel in Hz. If omitted, 1 Hz is set.

Returns 0 if all went well, or a negative error code.

See also:SampleEvent(), SampleTextMark(), SampleTitle$()

SampleLimitSize()

This corresponds to *Starting and stopping: Stop at file size* in the Automation 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 size is returned negated.

See also:SampleClear(), SampleLimitTime(), SampleMode(), SampleRepeats()

SampleLimitTime()

This corresponds to *Starting and stopping: Stop at time* in the Automation dialog.

```
Func SampleLimitTime( {time} );
```

time   The time in seconds to set as a limit. A positive time sets the limit and enables it. A negative time sets the limit to the positive time, but disables the limit. A value of zero, or omitting the argument, leaves the time limit unchanged.

Returns The limit before the call. If the limit is disabled, the time is returned negated.

See also:SampleClear(), SampleLimitSize(), SampleMode(), SampleRepeats()

SampleMode()

This function sets and gets the sampling mode in the Sampling configuration dialog. To set Triggered mode you should use SampleTrigger() as mode 3 is maintained here for backwards compatibility with previous versions of Spike2.

```
Func SampleMode(mode%{, time, trig%|every});
```

mode%   This argument determines the action of the command:
        0   Returns the current mode as 1, 2, or 3 for Continuous, Timed or Triggered. Any additional arguments are ignored.
        1   Sets Continuous recording mode.
        2   Sets Timed recording mode. All three arguments are required.
        3   Sets Triggered recording mode. All three arguments are required.
        -1   Returns the For field value, valid in Timed mode.
        -2   Returns the value of the Every field, valid for Timed sampling mode.
        -3   Returns the Trigger field value, valid in Triggered sampling mode.

time   In modes 2 and 3 it sets the For field of the Sampling mode group, in seconds.

every   In mode 2 it sets the Every field for timed sampling, in seconds.

trig%   The trigger channel number for mode 3. This channel must exist in the sampling configuration and can be of any type except a waveform or WaveMark channel.

Returns For modes 0-3 it returns the current sampling mode as 1-3, or a negative error code. In mode -1 to -3, it returns the information described for mode%, above.

If you set triggered mode, triggers 2 to 4 are disabled and the command is equivalent to calling `SampleTrigger(1, trig%, -1, 0.0, time, -1)`.

See also: `SampleClear()`, `SampleTrigger()`

SampleOptimise()

This function sets and gets the sample rate optimising settings in the Resolution tab of the Sampling configuration dialog. Spike2 can match the requested sample rates for waveform and WaveMark channels by changing the microseconds per time unit, time units per ADC convert and by making waveform channels into Quick, Slow or Copy channels. See the *Sampling data* chapter of the *Spike2 manual* for full details.

The available ADC sample rate is divided by giving one share to each WaveMark channel, one to each Quick channel and one to the entire Slow channel group. The actual rates for each waveform channel can then be further down-sampled by taking one point in n. For example, with 4 Slow channels, the fastest rate for a Slow channel is one quarter the fastest rate for a Quick channel.

The rates are optimised each time you change any aspect of the sampling configuration that affects the sample rate. With full optimise set and slow sampling rates, it could take an appreciable time for Spike2 to search all possible combinations for the best sample rates. For this reason, `SampleClear()` sets no optimise and version 3 compatibility. We recommend that you use `SampleOptimise()` after the other sample setup commands so that you pay the time penalty for optimising once.

```
Func SampleOptimise(opt%,grp%,type%{,usL%{,usH%{,dis%{,burst%}}}});
Func SampleOptimise(get%);
```

opt%   This sets the optimise method. Use -1 for no change or:
       0   No optimise of microseconds per time or time per ADC.
       1   Partial, optimise time per ADC, no optimise of microseconds per time.
       2   Full, optimise both time per ADC and microseconds per time.

grp%   This controls the use of Quick channels. Use -1 for no change or:
       0   Version 3 compatible; no Quick channels and channel divides up to 65535.
       1   Group channels with the same ideal rate so they get the same actual rate.
       2   Optimise for the least error in rate; channels with the same ideal rate may get different actual rates.

type%   Set the type of 1401 to optimise for or -1 for no change:
       0 Works with all supported 1401s except a 1401*plus* with an old ADC.
       1 1401*plus* with an old ADC.   4 Micro1401 mk II   7 Power1401 mk II
       2 Power1401   5 Power1401 625   8 Micro1401-3
       3 1401plus or a micro1401   6 micro1401 mk I   9 Power1401-3

usL%   If present, it sets the low limit, in microseconds, for optimising microseconds per time unit (when `opt%` is 2). Values outside the range 1 to 1000 are ignored.

usH%   If present, it sets the high limit, in microseconds, for optimising microseconds per time unit. Values outside the range 1 to 1000 are ignored.

dis%   A set of flags to disable features. Set -1 or omit for no change, else the sum of: 1=No 10 MHz clock, 2=No WaveMark copy, 4=No WaveMark divide, 8=No dummy channels.

burst%   Set -1 or omit for no change. Set 1 for burst mode, 0 for non-burst mode.

get%   Use with one argument to read back the current settings. The values 0 to 6 read back the values of `opt%, grp%, type%, usL%, usH%, dis%` and `burst%`.

Returns   The setting version returns the current optimise method. When called with one argument, the return values are as documented for `get%`.

See also: `SampleClear()`, `SampleLimitSize()`, `SampleLimitTime()`, `SampleTimePerAdc()`, `SampleUsPerTime()`

SampleRepeats()

This gets and optionally sets the Sampling configuration Automation tab Repeats field, which is used to sample a sequence of files with automatically incrementing names. You can use it online to return the number of files left to sample. You can choose to trigger each file in the sequence, or just the first with `SampleStartTrigger()`.

```
Func SampleRepeats({files%});
```

`files%` The number of files to sample; 0 and 1 both sample a single file. Set -1 to return the number of files that remain to be sampled (including the current file). If you omit this argument, the number of files is not changed.

Returns The number of repeats set at the time of the call or the number of files that remain to be sampled.

You must also set a file name with `SampleAutoName$()`, set automatic file saving with `SampleAutoFile(1)` and set a time or size limit with `SampleLimitTime()` or `SampleLimitSize()`, or load a suitable sampling configuration with these values set.

All files in the sequence have the same view handle. As each file reaches the time or size limit, it is saved, closed and replaced with the next file. The view handle always references a file, but its file name and sampling status will vary. The last sampled file in the sequence is not closed. You can use `SampleStop()` or `SampleAbort()` to end a sequence early and `SampleReset()` to restart a file.

See also: `SampleAutoFile()`, `SampleAutoName$()`, `SampleLimitSize()`, `SampleLimitTime()`, `SampleStartTrigger()`, `SampleStop()`

SampleReset()

This abandons sampling, deletes any data that has been written to disk, and returns to a state as if `FileNew()` had just been used to create a new data file. If it is used when sampling a sequence of files, the current file is restarted and the sequence continues.

```
Func SampleReset();
```

Returns 0 if the reset operation completed without a problem, or a negative error code.

See also: `SampleAbort()`, `SampleKey()`, `SampleStart()`, `SampleStop()`, `SampleStatus()`, `SampleText()`, `SampleWrite()`

SampleSeqClock()

This command changes the output sequence tick rate. You can change the default rate used when compiling a sequence that does not contain a `SET` or `SCLK` directive. You can also dynamically change the speed of a running sequence (the sequencer tick period is not guaranteed to be accurate during the change). When you change the default tick, time periods set in a sequence by `s()`, `ms()` and `us()` are set as accurately as the tick rate allows. If you change the rate of a running sequence, all sequencer-tick related periods, including periods set by `s()`, `ms()` and `us()`, are changed.

```
Func SampleSeqClock({which%{, ms}});
```

`which%` Set to 0 (or omit) to get or set the default tick period used when compiling a sequence. Set to 1 to get or set the tick period in use by a running sequence.

`ms` If present and greater than 0, this sets the new tick period in milliseconds.

Returns The tick period in milliseconds at the time of the call. If you set `which%` to 1 when there is no running sequence, the returned value is 0.

See also: `SampleKey()`, `SampleSeqStep()`, `SampleSequencer$()`, `SampleStart()`

SampleSeqCtrl()

This gets and optionally sets output sequence control values. See the Sampling configuration dialog Sequencer tab documentation for more information.

```
Func SampleSeqCtrl(opt%{, new%});
```

opt%   1   The sequencer jump control. new% values are: 0 = keyboard, control panel and script, 1 = control panel and script, 2 = script only.

      2   The minimum instructions to reserve space for in a sequence. Used when loading sequences during sampling.

      3   The minimum table space to reserve for dynamically loaded sequences.

new%   Optional. The new value for the control option selected by opt%.

Returns   The value of the item selected by opt% at the time of the call.

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()

This sets the sequencer file in the Sampling configuration, sets and gets the sequencer mode (None, Text or Graphical) and changes the sequence when sampling. To get the name of the current sequence file, use SampleSequencer$(). There are two variants:

```
Func SampleSequencer(new$);
Func SampleSequencer({mode%{, new$}});
```

new$ In the first variant this sets the name of a sequencer file; pass an empty string for no file. In the second variant omitting new$ is equivalent to passing an empty string. Setting an input file name that doesn't exist returns a negative error.

mode%   Omit or set -1 to return the sequencer mode. Set 0 for no sequencer, 1 to set a text sequencer using the name passed in new$, 2 to set a graphical sequence. Set 3 to write the current graphical sequence for the 1401 type set in the resolution

tab or by the `SampleOptimise()` command to the file name in `new$`, or 4 to change the current sequence during sampling.

Returns The first variant and the second with `mode%` in range -1 to 2 return a negative error or the sequencer mode at the time of the call as 0, 1 or 2. Mode 3 returns: 1=no graphical sequence, 0=success, -1=file could not be created, -2=write error, -3=segment has more than 10 varying pulses. Mode 4 returns: 0=success, 1=no sequence running, 2=not sampling, 3=internal error, 4=sequence too long, 5=table too long, or a negative code for a bad sequence file.

See also: `SampleKey()`, `SampleOptimise()`, `SampleSequencer$()`

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. From 4.04, values set before `SampleStart()` set the initial variable values. Previously they took effect after sampling started.

```
Func SampleSeqVar(sVar%{, new%});
```

`sVar%`   The sequencer variable to set or read, in the range 1 to 256 (64 with 1401*plus*).

`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 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()`

SampleStart()

This function can be used after `FileNew()` has created a new time view based on the current Sampling configuration. It starts sampling immediately, or on a trigger.

```
Func SampleStart({trig%});
```

`trig%`   0 = start sampling immediately (default), 1 = wait for a Trigger input signal (1401*plus* Event 3 input), -1 = use the sample configuration trigger setting.

Returns 0 if all went well or a negative error code.

See also: `SampleAbort()`, `SampleRepeats()`, `SampleReset()`, `SampleStop()`, `SampleStartTrigger()`, `SampleStatus()`, `SampleWrite()`

SampleStartTrigger()

This gets and optionally sets the trigger option used when sampling starts. It is equivalent to *Starting and stopping: Triggering* in the Sampling configuration Automation tab.

```
Func SampleStartTrigger({trig%});
```

`trig%`   Omit this argument for no change. The following values can be used:

-1   Use whatever state the user previously set in the sampling control panel
0   Sampling is not triggered
1   Sampling starts on a trigger; repeated files are not triggered
2   Sampling starts on a trigger; repeated files are triggered

Returns The trigger state (as for the `trig%` argument) at the time of the call.

See also:`SampleAbort()`,`SampleRepeats()`,`SampleReset()`,`SampleStop()`,
`SampleStart()`,`SampleStatus()`,`SampleWrite()`

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 time view is ready to sample, but it has not been told to start yet
1   Sampling is waiting for an Event 3 trigger
2   Sampling is now in progress
3   Sampling is stopping (the code changes to -1 when it has stopped)

See also:`SampleAbort()`,`SampleReset()`,`SampleStart()`,`SampleStop()`

SampleStop()

This function stops sampling in progress, and is equivalent to the Stop button of the floating command window. The function does not return until sampling has stopped. If used in a file sequence, the sequence is cancelled and the current file is saved.

```
Func SampleStop();
```

Returns 0 if sampling stopped correctly or a negative error code.

See also:`SampleAbort()`,`SampleKey()`,`SampleReset()`,
`SampleStart()`,`SampleStatus()`,`SampleText()`,`SampleWrite()`

SampleTalker()

This function adds a Talker channel to the Sampling Configuration. If the channel is already in use, it is replaced. See the Sampling Configuration dialog for Talker details.

```
Func SampleTalker(chan%, tkr$, port%);
```

chan%   The channel number to use for the new channel.

tkr$ The name of the Talker (this is not case sensitive).

port%   The data channel (defined by the Talker) to add. The first channel is 0.

Returns 0 if all went well, or a negative error code.

SampleText()

This function adds text to a text marker channel during sampling. Text marker channels are created with the `SampleTextMark()` command.

```
Func SampleText(text$ {,time {,code%[]}});
```

text$ The text string to attach to the text marker. If the string is longer than the maximum length set for the channel, extra characters are ignored.

Time   The time for the text marker. If this argument is omitted, negative, less than the time of the last text marker, greater than the current sampling time or this is the triggered sampling mode trigger channel, then the current sampling time is used

code% The first four elements of this array set the marker codes stored with the text string. If this argument is omitted the codes are set to 0. Codes are limited to the range 0-255. Only the lower 8 bits of codes outside this range are stored.

Returns 0 if all was OK, or a negative error code.

See also:SampleAbort(), SampleKey(), SampleReset(), SampleStart(), SampleStop(), SampleStatus(), SampleTextMark()

SampleTextMark()

This function sets channel 30 in the sampling configuration as a text marker or reads back the TextMark channel settings. Each event on a text marker channel holds a time, marker codes and a text string. You can add text markers to this channel using the SampleText() command and from a serial line. The channel title and comment are not changed; you can set these with SampleTitle$() and SampleComment$(). The first version of the command sets information, the second returns serial line information from the sampling configuration.

```
Func SampleTextMark(max%{, port%{, term$
                   {, baud%{, bits%{, par%{, stop%{, hsk%}}}}}}});
Func SampleTextMark({&term${,
                   &baud%{ ,&bits%{, &par%{, &stop%{, &hsk%}}}}}});
```

max%   The maximum number of characters that can be attached to each text marker in the range 1 to 200. If 0 is passed, the channel is deleted from the list.

port% Serial port, in the range 1 to 19, to use to read on-line TextMark data. If omitted, no serial data is read. Read this back with SampleChanInfo(). A read back port value of 0 means no serial line input.

term$ The terminating character for serial line read. If this is omitted when setting up the serial line, "\r" is used (carriage return, character code 13). Characters with codes less than 32 are ignored on input unless they are the terminator character.

baud% The serial line Baud rate (number of bits per second). The maximum character transfer rate is about one-tenth this figure. All standard Baud rates from 50 to 115200 are supported. If you do not set a Baud rate, 9600 is used.

bits% The number of data bits to encode a character (7 or 8). If omitted, 8 is set.

par%   Parity check: 0 for no parity check, 1 for odd parity or 2 for even parity. If you do not set parity, the default is no parity.

stop% The number of stop bits as 1 or 2. If not set, the default is 1 stop bit.

hsk%   The handshake (flow control) mode. 0=no handshake, 1=hardware handshake, 2=XON/XOFF protocol. If not set, the default is no handshake.

Returns The first version returns 0 or a negative error code, the second returns the maximum number of characters in each marker or 0 if the channel is not used (in which case the arguments are not changed).

See also:SampleText(), SampleChanInfo(), SampleComment$(), SampleClear()

SampleTimePerAdc()

This sets and gets the number of time units set by SampleUsPerTime() for each ADC conversion. The product of the time per ADC and the microseconds per unit time must not exceed the capabilities of your 1401 (use the sampling configuration dialog resolution tab to investigate allowed values). If the optimise method is set by SampleOptimise() is not 0, this call will have no effect as time per ADC will be recalculated.

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

new%   The number of clock ticks per conversion in the range 1 to 32767. If this is omitted the value is not changed. Illegal values stop the script with a fatal error.

Returns The value of the time per ADC convert at the time of the call.

See also:SampleOptimise(),SampleUsPerTime(),SampleWaveform()

SampleTitle$()

This gets and sets the title attached to a channel in the Sampling Configuration dialog.

```
Func SampleTitle$(chan% {,new$});
```

chan% The channel number.

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:SampleCalibrate(),SampleComment$(),SampleClear()

SampleTrigger()

This sets the four triggers that can be set to determine which data for a channel is written to disk. Each trigger is associated with a list of channels. The first trigger is also set by SampleMode() for backwards compatibility. You cannot change the triggering after sampling has started. Channels not associated with a trigger are written continuously.

```
Func SampleTrigger(trig%, scr%, code%, relS, relE{, cSpc});
Func SampleTrigger(trig%, get%);
```

trig% A trigger number in the range 1 to 4. Setting a trigger sets triggered sampling mode. This is cancelled by SampleClear() or by calling SampleMode().

scr% A source channel for triggers. The channel is not checked here, so you can put in anything that could be a channel number. However, it is checked when you start sampling and if this is not an event-based channel, the trigger is ignored.

code% If the src% channel is Marker-based, you can choose to only trigger when the first marker code matches this value (in the range 0 to 255). Set -1 to accept all markers. If the source is not a marker, code% is ignored.

relS The start of the triggered area relative to the trigger time, in seconds. This can be negative or positive, but must be less than relE. There are limits on how far into the past a trigger can go; there is a warning message when sampling starts if the triggering requests are impossible.

relE The end of the triggered area relative to the trigger time. This must be greater than relS. Spike2 saves data in buffers so you may get more data saved to disk that you requested.

cSpc A channel specification for the channels to trigger. If you omit this argument, all channels are selected for triggering. You cannot use the standard channel specification values -2 or -3.

get% You can use the second command variant to return trigger values. Get values:

-1	-2	-3	-4	-5
scr%	code%	relS	relE	CSpc, -1=all, 1-n for one, 0=multiple

If more than one channel is selected, the channel specification returns 0.

Returns The first variant returns 0 or an error code. The second variant returns the requested value.

See also:SampleClear(), SampleMode()

SampleUsPerTime()

This gets and optionally sets the basic time unit used for sampling. If the optimise method is set to full by SampleOptimise(2,...), changes to the basic time units have no effect as Spike2 chooses the best value to optimise the waveform sample rates.

```
Func SampleUsPerTime({new});
```

new   If present, this sets the basic time unit in the range 2-1000 microseconds. Out of
         range values cause a fatal script error. If you sample with a Power1401 or
         Micro1401 mk II or -3, the value is rounded to the nearest 0.1 microseconds.
         For all older 1401 types, the value is rounded to the nearest microsecond.

Returns The current value of microseconds per time unit.

See also:SampleLimitTime(), SampleOptimise(), SampleTimePerAdc()

SampleWaveform()

This function adds a waveform channel to the list of channels required. If the channel is
already in use, it is replaced. The units, scale and offset fields are set to the Spike2
defaults (input in Volts); use SampleCalibrate() to modify them. The channel title
and comment are not changed; you can set these with SampleTitle$() and
SampleComment$(). See the Sampling Configuration dialog for sample rate details.

```
Func SampleWaveform(chan%, port%, ideal);
```

chan%   The channel number to use for the new channel in the range 1 to 100.

port%   An unused (by a waveform channel) 1401 waveform port in the range 0-63.

ideal   The ideal sampling rate that you would like for the port in Hz. Remember that
         you may not get this rate. Spike2 will set the nearest rate it can.

Returns 0 if all went well, or a negative error code.

See also:SampleOptimise(), SampleTimePerAdc(), SampleUsPerTime()

SampleWaveMark()

This command adds a WaveMark channel to the sampling configuration and returns
information about a WaveMark channel. The units, scale and offset fields are set to the
Spike2 defaults (input in Volts); use SampleCalibrate() to modify them. The channel
title and comment are not changed; you can set these with SampleTitle$() and
SampleComment$(). See the Sampling Configuration dialog for sample rate details. The
first command version sets the channel, the second returns information:

```
Func SampleWaveMark(chan%, port%, rate,size%,pre%{,ideal{,nTr%}});
Func SampleWaveMark(chan%{, &pre%{, &nTr%}});
```

chan%   The channel number of the WaveMark channel in the range 1 to 100.

port%   The 1401 waveform port in the range 0-63.

rate   The estimated maximum sustained spike rate, used to allocate buffer space.

size%   The number of waveform samples per trace to save for each WaveMark. This
         must be an even number in the range 6 to 126.

pre%   The number of points before the peak/trough of each spike, range 0 to size%-1;

ideal   If present, this sets the ideal sample rate, in samples per second, for all
         WaveMark channels. This must be in the range 0 to 500000.0 Hz. Rates of 1 Hz
         or less are ignored and do not change the WaveMark rate.

nTr%   If present, this sets the number of traces to sample as 1, 2 or 4. The default is 1.

Returns The setup call returns 0 for success or a negative error code. The get information
         call returns size% (points per trace) or 0 if this is not a WaveMark channel.

You can use SampleChanInfo() to read back the port%, rate and ideal fields.

See also:SampleOptimise(), SampleTimePerAdc(), SampleUsPerTime()

SampleWrite()

This controls writing data to the file during sampling. You can enable and disable writing of some or all channels, and obtain the sampling state of channels and channel groups.

```
Func SampleWrite(write% {,chan%|chan%[]});
```

write% This determines the action taken by the command:
- -1 Report on the state of the channel (or channels) given by the next argument.
- 0 Disable writing to disk (pause) the channels given by the next argument
- 1 Enable writing to disk (un-pause) the channels given by the next argument

chan% This sets the channels to operate on. If omitted, all channels are used. If this is an integer, it is the channel number. If it is an array, index 0 holds the number of channels following; the remaining elements hold a list of channel numbers.

Returns The state of writing for the channel or channels:
- 0 Disabled   1 Enabled   2 Some channels in list are enabled

See also:SampleAbort(), SampleKey(), SampleReset(), SampleStart(),
    SampleStop(), SampleStatus(), SampleText()

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 .s2s, 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.s2s|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, taken as 0 if omitted. Sum of: 1 = run even if the current script ends in an error, 2 = keep loaded script in memory.

If the file you name does not exist when Spike2 tries to run it, nothing happens. If the nominated script is not already loaded, Spike2 will load it, run it and unload it unless the keep loaded script in memory flag is set. If a loaded script calls Yield() or calls any function that allows the system to idle (Toolbar(), DlgShow()...), the script can be unloaded while it is still running. This is usually harmless unless the loaded script attempts to use App(3), which will return 0 if the script is no longer in memory.

See also:App(), Profile()

Seconds()

This returns the time in seconds for use in relative time measurements. You can also set the timer. Use `Maxtime()` for time into sampling. The online help has more information.

```
Func Seconds({set{, hiRes}});
```

set   If present, this sets the time in seconds. The time is 0 when Spike2 starts.

hiRes   If present, 0 (default) is 1 ms resolution, 1 for the highest available resolution.

Returns The time in seconds or the timer resolution if the `hiRes` argument is present.

See also:`Date$()`, `MaxTime()`, `Time$()`, `TimeDate()`

Selection$()

This function returns the text in the current view that is currently selected.

```
Func Selection$();
```

Returns The current text selection. If no text is selected, or if the view is inappropriate for this action, an empty string is returned.

See also:`EditCopy()`, `EditCut()`, `EditPaste()`, `MoveBy()`, `MoveTo()`

SerialClose()

This function closes a serial port opened by `SerialOpen()`. Closing a port releases memory and system resources and will throw away any characters in the output buffer. 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 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 rate limits.

```
Func SerialOpen(port%{, baud%{, bits%{, par%{, stop%{, hsk%}}}}});
```

port%   The serial port to use, in the range 1 to 256. The number of ports depends on the computer.

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 omit baud%, 9600 is used.

bits%   The number of data bits to encode a character. Windows supports 4 to 8 bits, the Macintosh supports 7 or 8. If bits% is omitted, 8 is set. 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. To use an array of strings you must set a terminator or 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 and 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% or in%[]) when it 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 integer array size, the size of the buffers used by Spike2 to process data and by the size of the system buffers used outside Spike2. 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. Use SerialCount() to test for items to read to avoid a time out.

See also:SerialOpen(), SerialWrite(), SerialCount(), SerialClose()

SerialWrite()

This writes one or more strings or binary data to a serial port opened by `SerialOpen()`. Call with a single argument to return the available output buffer space in characters.

```
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. The return value is the number of strings written to the output.

out%   A single integer or an integer array to write as binary. One value is written per integer. The output written 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`. The return value is 1 if the transfer succeeded.

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 For success, the return value is as documented for the `out` argument. If there is no room in the output buffer for the data the return value is -1 except when `out$[]` is used when the return values is the count of strings actually sent. With a single argument, the return value is the space in the output buffer.

See also:`SerialOpen()`, `SerialRead()`, `SerialCount()`, `SerialClose()`

SetXXXX commands

These commands create result views attached to the current time view or to the time view associated with the current result view. Apart from `SetResult()`, which creates a result view that is not dependent on a time view, these routines match the Analysis menu New Result view commands. They do not update the display (use `Draw()` or `DrawAll()`), nor do they perform any analysis (see the `Process()` command).

The commands return a view handle, or a negative error code. Errors include: Bad channel number, illegal number of bins, out of memory, illegal bin size. The new view is made the current view. However, it is created invisibly and must be made visible with `WindowVisible(1)` before it will appear when drawn.

Most commands accept a channel list specifier, written as `cSpc`. It can be an integer channel number, or -1 for all, -2 for visible or -3 for selected channels. It can also be a channel specification string, such as `"1..4,6,10"` or an integer array where the first element is the channel count, the remaining elements are the channel numbers. Whatever channels are specified, only channels of the correct type for the command are used. If a channel is included twice, the first occurrence is used. It is an error if the resulting list is empty. Result view channels appear in the same order as in the specification. The first channel in `cSpc` generates result channel 1, the second generates channel 2, and so on.

SetAverage()

This command creates a result view to hold the sum or average of sweeps of waveform data. The `Process()` command does the analysis. `Sweeps()` reports the number of sweeps accumulated. Omitted optional arguments have the value 0. The function is:

```
Func SetAverage(cSpc, bins% {,offset {,trig% {,flags%{,align%}}});
```

cSpc   A channel list specifier of waveform or WaveMark channels to average in the current time view, or in the time view associated with the current result view. Invalid channels are ignored. The channel list must hold at least one valid channel. All the channels must have the same sampling rate; any channels that do not match the sample rate of the first channel in the list are ignored.

bins%   The number of bins required. There must be at least one bin. The maximum is limited by available memory. The bin width is the waveform sampling interval.

offset   This sets the pre-trigger time to show in the result. If omitted, 0 is used.

trig%   The channel number to use as a trigger for each sweep. If this is omitted, or set to 0, then each call to `Process()` takes the start time as the trigger time.

flags%   This is the sum of flag values: 1=display the mean data and not the sum, 4=enable error bars, 32=count items per bin. If `flags%` is omitted, 0 is used.

align%   How to align data to the trigger. 0=Next point, 1=nearest point, 2=linear interpolation, 3=cubic spline interpolation. If omitted, 0 is used.

Returns   The function returns a handle for the new view, or a negative error code.

See also:`BinError()`, `ChanData()`, `DrawMode()`, `Process()`, `Sweeps()`

SetEvtCrl()

This creates a result view of event correlation histograms and optional raster displays. The `Process()` command does the analysis. `Sweeps()` reports the number of triggers processed. You can take measurements from an auxiliary channel from each sweep and use this value to sort rasters, display a symbol and/or discard sweeps. See the Event correlation in the Analysis menu for details. See `RasterSort()` and `RasterSymbol()` for a description of sort values and symbol times.

```
Func SetEvtCrl(cSpc, bins%, binsz{, offset{, trig%{, flags%
                                        {, aCh%{, Mn, Mx}}}}});
```

cSpc   A channel list specifier of event or marker channels in the current time view, or in the time view associated with the current result view. Invalid channels are ignored. The resulting channel list must hold at least one valid channel.

bins%   The number of bins in the histogram. There must be at least 1 bin.

binsz   The width of each bin in seconds. This is converted into underlying time units.

offset   This sets the pre-trigger time to show in the histogram.

trig%   The channel number to use as a trigger for each sweep. If this is omitted, or set to 0, then each call to `Process()` takes the start time as the trigger time.

flags%   This is the sum of: 1 to scale the result as spikes per second, 2 to enable raster displays, 8 for backwards `aCh%` event search, 16 to exclude rather than include values that lie in the `Mn` to `Mx` range. If omitted, 0 is used.

aCh%   Auxiliary measurement channel. The measured value sets raster sort value 1. For waveform or RealWave channels, this is the waveform value at the trigger time. For all other channel types, it is the latency of the first event on the channel before/after the trigger and it also sets the time of symbol 1. If you omit `aCh%`, there is no auxiliary channel.

Mn,Mx   If present, these arguments set the range of measured values from `aCh%` that control if a sweep is included or excluded.

Returns   The function returns a handle for the new view, or a negative error code.

For an auto-correlation, set `trig%` the same as `chan%`. In this case, we do not count the result of correlating each event with itself. If you must count self-correlations, add `Sweeps()` to the bin holding the time shift of zero.

See also:`SetEvtCrlShift()`, `DrawMode()`, `Process()`, `RasterSet()`, `Sweeps()`

SetEvtCrlShift()

This can be used when the current result view was created by `SetEvtCrl()`. It sets a time shift for the next `Process()` command to produce shuffled correlations by shifting one channel with respect to the other without changing the result view time axis.

```
Proc SetEvtCrlShift(shift);
```

shift   This value can be positive or negative. It causes the data on the `trig%` channel to be correlated with data from channel `chan%` at a time `shift` later. The shift is set back to zero after the `Process()` command on this result view.

Consider two channels of events (to be correlated) and a channel marking stimuli:
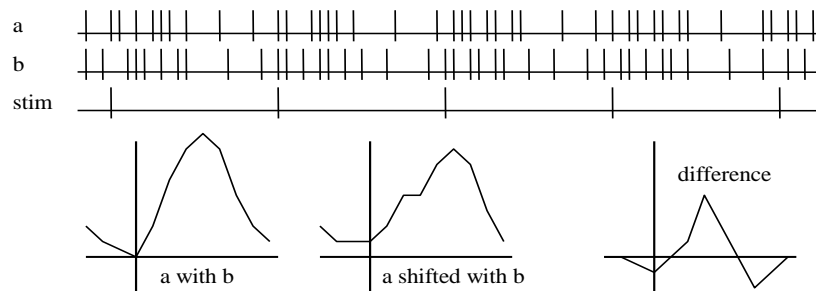
If the stimulus channel holds events at some constant interval `int`, by shifting all events on one of the two channels by `int`, the only correlation between the two channels left is the correlation due to the stimulus. Thus by forming the difference between the unshifted correlation and the shifted correlation, you can obtain the correlation between a and b with the effect of the stimulus removed.

See also:`SetEvtCrl()`, `Process()`

SetINTH()

This function creates a result window to hold an interval histogram. The `Process()` command does the analysis. Each interval processed increments the value returned by `Sweeps()`, even intervals that are too short or too long to contribute to the histogram.

```
Func SetINTH(cSpc, bins%, binsz{, minInt});
```

cSpc   A channel list specifier of event or marker channels in the current time view, or in the time view associated with the current result view. Invalid channels are ignored. The resulting channel list must hold at least one valid channel.

bins%   The number of bins required. There must be at least one bin. The maximum is limited by available memory.

binsz   The width of each bin in seconds. This is converted into underlying time units.

minInt This sets the start of the first analysis bin, in seconds. Shorter intervals are counted for `Sweeps()`, but not included in the histogram. If omitted, 0 is used.

Returns The function returns a handle for the new view, or a negative error code.

See also:`Process()`, `Sweeps()`

SetPhase()

This function creates a result view to hold phase histograms with optional raster displays. The `Process()` command does the analysis. `Sweeps()` reports the number of cycles analysed. You can measure a value for each sweep from an auxiliary channel and use this value to sort rasters, display a symbol and discard sweeps. See the description of the Analysis menu Phase histogram for details. See `RasterSort()` and `RasterSymbol()` for a description of sort values and symbol times.

```
Func SetPhase(cSpc, bins%{, minCyc{, maxCyc{, cycle%{, flags%
                                        {, aCh%{, Mn, Mx}}}}}});
```

cSpc   A channel list specifier of event or marker channels in the current time view, or in the time view associated with the current result view. Invalid channels are ignored. The resulting channel list must hold at least one valid channel.

bins%   The number of bins required. There must be at least one bin. The maximum is
        limited by available memory. The bins are each 360/bins% wide.

minCyc The minimum cycle time to use in seconds. If two consecutive cycle markers are
        closer than this time, the cycle is ignored, and is not counted by Sweeps(). If
        this is omitted, a value of 0.0 is used.

maxCyc The maximum cycle time to use in seconds. If two consecutive cycle markers
        are further apart than this, the cycle is ignored, and is not counted by Sweeps().
        If this is omitted, there is no limit on cycle size.

cycle% The channel number to use as the cycle start and end marker. If this is omitted,
        or set to 0, then each call to Process() takes the start time and end time as
        marking a single cycle.

flags% This is the sum of: 2 to enable raster displays, 8 for backwards aCh% event
        search, 16 to exclude rather than include sweeps with measured values that lie in
        the Mn to Mx range. If omitted, 0 is used.

aCh%   Auxiliary measurement channel. The measured value sets raster sort value 1. For
        waveform or RealWave channels, this is the waveform value at the cycle start.
        For all other channel types, it is the latency of the first event on the channel
        before/after the cycle start and it also sets the symbol 1 time. If you omit aCh%,
        there is no auxiliary channel.

Mn,Mx   If present, these arguments set the range of measured values from aCh% that
        control if a sweep is included or excluded.

Returns The function returns a handle for the new view, or a negative error code.

See also:DrawMode(), SetEvtCrl(), Process(), RasterSet(), Sweeps()

SetPower()

This function creates a result view to hold a power spectrum. The Process() command
does the analysis. The function is as follows:

```
Func SetPower(cSpc, fftsz% {,wnd%});
```

cSpc   A channel list specifier of waveform channels in the current time view, or in the
        time view associated with the current result view. Invalid channels are removed
        from the list. The resulting channel list must hold at least one valid channel. All
        the channels must have the same sampling rate; any channels that do not match
        the sample rate of the first channel in the list are ignored.

fftsz% The size of the transform used in the FFT. This must be a power of 2 in the
        range 16 to 16384. The result 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().

wnd%   The window to use. 0 = none, 1 = Hanning, 2 = Hamming. Values 3 to 9 set
        Kaiser windows with -30 dB to -90 dB sideband ripple in steps of 10 dB. If this
        is omitted a Hanning window is applied.

Returns The function returns a handle for the new view, or a negative error code.

See also:ArrFFT(), Process(), Sweeps()

SetPSTH()

This creates a result view to hold a peri-stimulus time histogram with an optional raster
display. The Process() command does the analysis. Sweeps() returns the number of
triggers processed. You can measure a value for each sweep from an auxiliary channel
and use this to sort rasters, display a symbol and discard sweeps. See the description of
the PSTH in the Analysis menu for details. See RasterSort() and RasterSymbol()
for a description of sort values and symbol times.

```
Func SetPSTH(cSpc, bins%, binsz {,offset {,trig% {,flags%
                                          {,aCh% {,Mn,Mx}}}}});
```

cSpc   A channel list specifier of event or marker channels in the current time view, or
        in the time view associated with the current result view. Invalid channels are
        ignored. The resulting channel list must hold at least one valid channel.

bins%   The number of bins required. There must be at least one bin.

binsz   The width of each bin in seconds. This is converted into underlying time units.

offset This sets the pre-trigger time, in seconds. If omitted, 0.0 is used.

trig%   The channel number to use as a trigger for each sweep. If this is omitted, or set
        to 0, then each call to Process() takes the start time as the trigger time.

flags% This is the sum of: 1 to scale the result as spikes per second, 2 to enable raster
        displays, 8 for backwards aCh% event search, 16 to exclude rather than include
        values that lie in the Mn to Mx range. If omitted, 0 is used.

aCh%   Auxiliary measurement channel. The measured value sets raster sort value 1. For
        waveform or RealWave channels, this is the waveform value at the trigger time.
        For all other channel types, the value is the latency of the first event on the
        channel before/after the trigger and it also sets the symbol 1 time. If you omit
        aCh%, there is no auxiliary channel.

Mn,Mx   If present, these arguments set the range of measured values from aCh% that
        control if a sweep is included or excluded.

Returns The function returns a handle for the new view, or a negative error code.

See also:DrawMode(), SetEvtCrl(), Process(), RasterAux(), Sweeps()

SetResult()

This function creates a result view of user-defined type, attached to no time view and
with no implied Process(). Sweeps() will return 0 unless you set the sweep count.

```
Func SetResult({chans%,} bins%, binsz, offset, title$, xU$
                          {,yU$ {,xT$ {,yT$ {,flags% {,tick}}}}});
```

chans% This sets number of channels in the new result view. Omit for 1 channel. See the
        View() documentation for access to multiple channel result view data.

bins%   The number of bins in the view.

binsz   The width of each bin. Bins should have a positive non-zero width.

offset The x axis value at the start of the first bin.

title$ The 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).

flags% Add 1 for mean (not sum) mode for errors, 2 to enable raster data, 4 to enable
        error bars, 32 for individual counts for each bin. The default value is 0.

tick   This is required if you enable raster data. Result views store raster data as 32-bit
        integers. tick is the time resolution for this data. When working with a time
        view, set tick to BinSize(), the time resolution of the time view. The
        maximum time of a raster event is 2147483647*tick seconds.

Returns The function returns a handle for the new view, or a negative error code.

See also:BinError(), RasterGet(), RasterSet(), Sweeps(), View()

SetWaveCrl()

This creates a result view to hold the waveform correlation between waveform channels sampled at the same rate. The `Process()` command does the analysis. `Sweeps()` returns the number of data points used on the reference channel.

```
Func SetWaveCrl(cSpc, ref%, bins% {,offset});
```

cSpc   A channel list specifier of waveform channels in the current time view, or in the time view associated with the current result view. Invalid channels are removed from the list. There must be at least one valid channel. Channels that do not match the sample rate of the first channel in the list are ignored.

ref%   A reference waveform channel that is correlated with all the waveform channels.

bins%   The number of bins in the correlation. The bin width is the channel sampling interval. This calculation is slow compared to the other `SetXXXX` commands. The time taken is proportional to the length of the area processed times `bins%`.

offset   The time to show before the zero time shift, in seconds. If omitted, 0 is used.

Returns   The function returns a handle for the new view, or a negative error code.

See also: `SetWaveCrlDC()`, `Process()`, `Sweeps()`

SetWaveCrlDC()

To use this function the current view must be a waveform correlation result view. It sets whether the DC levels of the two signals is removed before the correlation or not.

```
Func SetWaveCrlDC({useDC%})
```

useDC%   If present, a zero value removes the DC, a non-zero value (default) includes it.

Returns   The `useDC%` value before the call. It returns a negative error if this view is a result view but not a waveform correlation.

If you change the setting, the view is drawn at the next opportunity.

See also: `SetWaveCrl()`, `Process()`

Sin()

This calculates the sine of an angle in radians, or converts an array of angles into 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 and returns 0 or a negative error code. When the argument is not an array the function returns the sine of the angle.

See also: `ATan()`, `Cos()`, `Cosh()`, `Sinh()`, `Tan()`, `Tanh()`

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: `ATan()`, `Cos()`, `Cosh()`, `Sinh()`, `Tan()`, `Tanh()`

SMControl()

This function sets and reads the toolbar and edit field states of the current spike monitor window. See the documentation of the spike monitor window for a full description.

```
Func SMControl(item%{, new});
```

item% This identifies the item:

0	Draw mode	0=3D, 1=2D, 2=2D separated
1	Fade to background	0=no fade, 1=fade
2	Colour last spike only	0=colour all, 1=colour last, the rest are grey
3	Use thick lines	0=use thin lines, 1=use thick lines (slow)
4	Timed (smooth) update	0=update on new data, 1=update on timer
5	At end mode	0=use cursor 0, 1=at end of file
6	Lock y axes	0=track data size, 1=fix at current size
7	Show duplicate channels	0=Show originals only, 1=show duplicates
8	Maximum spikes	In the range 1 to 40
9	Time range	In the range 0.003 to MaxTime()
10	Display rectangles	0=no, 1=show front and back rectangles
11	Front rectangle x offset	In the range 0 to 0.9
12	Front rectangle scale	In the range 0.1 to 1.0
13	Back rectangle x offset	In the range 0 to 0.9
14	Back rectangle scale	In the range 0.1 to 1.0

new   If present, this sets the state of the item. See the table for acceptable values.

Returns The item state at the time of the call or -1 if the item does not exist.

See also:SSChan(), SSClassify(), SSOpen(), SSParam(), SSRun()

SMOpen()

This function gets the spike monitor window handle or opens it. The current view must be a time view or a spike monitor window. Close the spike monitor window with FileClose(). Use the SMControl() command to control the window. You can use the Window() and WindowVisible() commands to size, show and hide the window.

```
Func SMOpen({type%{, mode%});
```

type% This argument is assumed to be zero if omitted. Allowed values are:

- 0   Returns the view handle of the spike monitor window associated with the current view, or 0 if there is no open window.
- 1   Open the spike monitor window. Returns the handle or 0 if we failed.

mode% If this is zero or omitted, the dialog is created invisibly. Set 1 to make it visible. This is ignored unless type% is 1.

Returns The return value is the view handle of the spike monitor window or 0.

See also:FileClose(), SMControl(), ViewLink(), Window(), WindowVisible()

Sound()

This command either plays a tone of set pitch and duration (a short "beep" in Windows 9x) or it plays a .WAV file or system sound if your system has multimedia support.

```
Func Sound(freq%, dur{, midi%});              Tone output (not using sound card)
Func Sound(name${, flags%});                  Multimedia sound card 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 present and non-zero, freq% is interpreted as a MIDI value.

name$ The name of a .wav file or of a system sound. You can 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.

A blank name halts sound output. If name$ is any of the following (case is important), a standard system sound plays:

"S*"	Asterisk	"SS"	Start	"SE"	Exit	"SH"	Hand
"SW"	Welcome	"S?"	Query	"SD"	Default	"S!"	Exclamation

flags% This optional argument controls how the data is played. It is the sum of:

0x0001 1 Play asynchronously (start output and return). Without this flag, Spike2 does nothing 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) returns 1 if a previous asynchronous sound is finished and 0 if not.

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. We do not provide 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 2006, 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 an array of real numbers. Negative numbers halt the script 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.

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

SS...() commands

This family of commands gives you script control over the Spike Shape dialog. You open the dialog, or get the handle to an already open dialog with the `SSOpen()` command. You get and set the current channel with `SSChan()` and step or run with the `SSRun()` command. Use `SSParam()` to access the template parameters dialog. The `SSTemp...()` command family gives you access to the templates. The `SSButton()` command lets you set and get button and checkbox states in the dialog that control how it operates. The `HCursor()` command controls the horizontal cursors. The `SSCol...()` commands give you script control over collision analysis. Use `FileClose()` to shut the dialog.

SSButton()

This function sets and reads the toolbar and checkbox states of the current spike shape window. See the documentation of the spike shape dialogs for a description of the items.

```
Func SSButton({item%{, new%}});
```

item%   If omitted, all items are reset to 0. Otherwise, this identifies the item:

0	Overdraw spikes	0=no overdraw, 1=overdraw
1	Show template boundaries	0=hide, 1=show
2	Show non-matching spikes	0=hide, 1=show
3	Play sound for each spike	0=no sound, 1=sound
4	Scroll time view to track cursor 0	0=no track, 1=track
5	At end mode (online)	0=not at end, 1=at end
6	Circular replay	0=stop at end, 1=circular
7	Make templates	0=no, 1=yes
8	Build by code	0=by shape, 1=by code
9	Number of horizontal cursors	0=2, 1=4
10	Set which marker code to use	0-3 for layers 0-3
11	Overlay drawing of traces	0=no overlay, 1=overlay
12	Set size of displayed templates	0=small, 1=medium, 2=large
13	Set collision analysis mode	0=no, 1=yes: Edit WaveMark only
14	Collision: mean width checkbox	0=no, 1=yes

15 Collision: split as ideal checkbox   0=no, 1=yes: memory channel only

new%   If present, this sets the state of the item. Use 1 to select the feature (depress the button or check the box) and 0 to unselect it. Item 10 supports values 0-3.

Returns The item state at the time of the call or -1 if the item does not exist or no item%.

See also:SSChan(), SSClassify(), SSOpen(), SSParam(), SSRun()

SSChan()

This function sets and gets the current channel in the current spike shape dialog.

```
Func SSChan({ch%});
```

ch%   If omitted or 0, the return value is the current channel. If greater than 0, ch% sets the channel and the return value is the ch% (or the original channel if channel ch% is not suitable). If the channel changes, the old channel settings are saved.

If ch% is -1, the return value is the number of interleaved traces in the dialog. In the New Stereotrode or New Tetrode dialogs, this is 2 or 4. In the Edit WaveMark dialog, it is the number of interleaved traces for the current channel.

If ch% is -2, this forces the channel configuration to be saved; the return value is 0. Use this before FileClose() as that does not save the current settings.

In a New Stereotrode or New Tetrode dialog ch% is a zero based channel index into the list of channels to combine and the return value is the channel number.

Returns The return value depends on the ch% argument.

See also:SSButton(), SSClassify(), SSOpen(), SSParam(), SSRun()

SSClassify()

This function is equivalent to the New Channel button in the New WaveMark dialog and to the Reclassify button in the Edit WaveMark dialog. The arguments from new% onward are ignored in the Edit WaveMark dialog.

```
Func SSClassify({speed%{, new%{, type%{, noQu%}}}});
```

speed%   If this is omitted, the function returns 1 if the dialog is currently reclassifying or creating a new channel and 0 if it is not. If present, use the values:
-1 To cancel any ongoing reclassification or writing to a new channel.
0   Start reclassifying or writing a new channel with display updates. This takes place in idle time, so you must provide some with Yield() or by using Toolbar(), Interact() or DlgShow().
1   Reclassify or write a new channel in fast mode, with no display updates. The operation will complete before control returns to the script.

new%   The channel number to create in the New WaveMark dialog.

type%   The channel type to create. You can use types: 2 (Event-), 3(Event+), 4(Level), 5(Marker) or 6 (WaveMark) only. If omitted, type 6 (WaveMark) is used.

noQu%   If the channel set by new% is already in use and this argument is omitted or 0, the user is asked if they want to overwrite it. Set non-zero for no query.

Returns If speed% is omitted, the return value is 1 if reclassification or writing a new channel with display updates is in progress, otherwise 0. Otherwise the return value is 1 for success and 0 for failure.

See also:SSButton(), SSChan(), SSOpen(), SSParam(), SSRun()

SSColApply()

This function can only we used with an Edit WaveMark dialog as the current view or you will get a "Wrong view type" error. It applies the result of collision analysis to the current spike in the dialog. The dialog must be in collision analysis mode.

```
Func SSColApply({act%});
```

act%   This optional variable determines the action to perform. If the value is omitted it is taken as 0. The possible values are:

0   Test if it is possible to split the waveforms (equivalent to using the Split spike button in the collision analysis dialog). This can only be done if the current channel of spikes is stored in a memory channel. The return value is 1 if splitting the current spike is possible, 0 if not.

1   Equivalent to the Copy codes button in the collision analysis dialog. The return value is the number of codes that were copied. This will normally be 2 or 1, being the number of templates that contributed to the best match.

2   Equivalent to the Split spike button. The return value is -1 if it is not possible to split the current spike, otherwise the return value is the number of spikes that the current spike is split into.

Returns   If the current view is an Edit WaveMark dialog, but it is not in collision analysis mode, the return value is -1. Otherwise, the return value is as described above.

See also:SSButton(), SSChan(), SSColInfo(), SSOpen(), SSParam(), SSRun()

SSColArea()

This function can only we used with an Edit WaveMark dialog as the current view or you will get a "Wrong view type" error. It only be used in collision analysis mode and gets and sets the *Important area* used for collision analysis. Note that this area is reset to lie within the template region each time the dialog enters collision analysis mode. The command has two variants:

```
Func SSColArea(&start%);                        Get the start and return the size
Func SSColArea(newSt%, newSz%);        Set the start and size, return original size
```

start%   This is an integer variable, that is returned holding the zero-based offset from the start of the displayed data to the collision analysis area.

newSt%   This sets the start offset of the collision analysis area.

newSz%   This sets the size of the area (in data points) used for collision analysis.

Returns   The return value is the original size of the area used for collision analysis, or -1 if the spike shape dialog is not in collision analysis mode.

See also:SSButton(), SSChan(), SSOpen(), SSParam(), SSRun()

SSColInfo()

This function can only we used with an Edit WaveMark dialog as the current view or you will get a "Wrong view type" error. The dialog must be in Collision Analysis mode. The function gets information about the current state of matching two templates to the displayed spike.

```
Func SSColInfo(&error{, &tmp1%{, &tmp2%{, &pos1{, &pos2}}}});
```

error   This is a real variable that is returned holding the mean error per point. This is mean ratio of the difference between the waveform and the waveform templates with respect to the template width (either point by point, or to the mean width over all templates, depending on the Mean width checkbox).

tmp1%   An integer variable that is returned holding the 0-based index of the template that matches furthest to the left.

tmp2%   An integer variable that is returned holding the 0-based index of the template that matches furthest to the right. If only one template matches, this is negative.

pos1   A real variable that is returned holding the point position in the display at which the first point of the first template starts. This will, in general, be a fractional position as the best match is arrived at by interpolation.

pos2   This returns the position of the second matching template in the same format as pos1 . If there is no second template, the value is set the same as pos1.

Returns   The number of matching templates (0, 1 or 2) or -1 if the spike shape dialog is not in collision analysis mode.

See also:SSButton(), SSChan(), SSColArea(), SSOpen(), SSParam(), SSRun()

SSOpen()

This function opens and returns information about spike shape dialogs. The current view must be a time view or a spike shape dialog. Close this dialog with FileClose().

```
Func SSOpen({type%{, mode%}});
```

type%   This argument is assumed to be zero if omitted. Allowed values are:

-1   The function returns the type of any open spike shape dialog associated with the current time view or 0 if there is no open dialog. Returned values are 1=Edit WaveMark, 2=New WaveMark, 3=New n-trode.

0   The return value is the window handle of any open spike shape dialog associated with the current view, or 0 if there is no open dialog.

1   Open the Edit WaveMark dialog and return its handle or 0.

2   Open the New WaveMark dialog and return its handle or 0.

3   Open the New n-trode dialog with selected channels. Return its handle or 0.

mode%   If this is zero or omitted, the dialog is created invisibly. Set 1 to make it visible. This is ignored unless type% is greater than 0.

Returns   The return value depends on the type% argument.

See also: SSChan(), SSClassify(), SSParam(), SSRun(), ViewLink()

SSParam()

This function sets the template parameters for the current spike shape dialog and reads back the state of individual parameters. This is equivalent to the template parameters dialog; see that dialog for a full description of the arguments. The command has two variants. In the first, you supply all the arguments (use the value -1 to leave an argument unchanged). In the second variant, you can get or set the value of a single item.

```
Func SSParam(new%,wide,rare%,AMax,PMin,flg%,mode%,nAT%,int%,tc%);
Func SSParam(item%{, value});
```

new%   If new% is greater than 0, this sets number of spikes for a new template.

wide   The width of a new template as a percentage of the template amplitude.

rare%   Templates with spikes rarer than 1 in rate% are discounted.

AMax   Maximum amplitude change for tracking or 0 for no amplitude tracking.

PMin   Minimum percentage of point in the template for a possible match.

flg%   This is the sum of: 1 = use minimum percentage only when building templates, 2 = Remove DC value from spike data before matching.

mode%   The mode for adding spikes to the template: 0=All, 1=Autofix, 2=track.

nAT%   The number of spikes for Autofix and track modes.

int%   The interpolation method to use: 0=linear, 1=parabolic, 2=cubic spline.

tc%   The high-pass time constant to apply to the data expressed as 2 to the power tc% data samples with tc% in the range 1 to 30. Use 31 for no high pass filter.

item% Used in the second command variant. Set -2 to copy the current settings to all channels. Set -1 to reset the values to a standard state. Set 0 to close the parameters dialog if it is open. Otherwise set 1 to 10 to select one of the arguments new% (1) to tc% (10) and the return value will be the value of that item at the time of the call.

value If present, this sets the new value of the parameter when item% is greater than 0.

Returns The first command variant returns 0. The second variant, with item% greater than 0, returns the item value at the time of the call and item% less than zero returns 0. For item%=0, the return value is 1 if the parameter dialog was open.

See also:SSButton(), SSChan(), SSClassify(), SSOpen(), SSRun()

SSRun()

This function gets and sets the run state of the current view, which must be a spike shape window. Use Cursor(0) to get the current run position.

```
Func SSRun({run%});
```

run% If omitted, no change is made, otherwise negative values are backwards and positive are forwards. 0 means stop, 1=step, 2=run as fast as possible, 3= run in real time, 4-= run at 91 Hz, 5=31 Hz and so on up to 15=1 Hz.

Returns The run state at the time of the call.

See also:SSButton(), SSChan(), SSClassify(), SSOpen(), SSParam()

SSTempDelete()

This function deletes one or more templates for the current channel of the current view, which must be a spike shape window. If this command is used with no arguments, or with both arguments set to -1, the entire template system is cleared, equivalent to clicking the clear all templates button in the dialog.

```
Func SSTempDelete({n%{, code%}});
```

n%   The zero-based index of a template or -1 for all templates. Templates that match this index and the code% argument are deleted. If this is omitted, -1 is used.

code% If this is omitted or set to -1, all templates codes are deleted. Otherwise, only templates selected by n% that match code% are deleted.

Returns The number of deleted templates or -1 if n% is not a template index.

See also:SSButton(), SSChan(), SSClassify(), SSOpen(), SSParam(), SSRun(), SSTempGet(), SSTempInfo(), SSTempSet(), SSTempSizeGet(), SSTempSizeSet()

SSTempGet()

This function gets information about the current set of templates for the current channel of the current view, which must be a spike shape window. It can also return the current raw data displayed in the template window.

```
Func SSTempGet(n%{, temp[]{[]}{, what%{, &count%}}});
```

n%   The zero-based index of the template to return information from or -1 to get the last triggered sweep or -2 for the displayed waveform (triggered or not). Use SSTempSizeGet() to get the number of displayed points and the number of points in the template. The -2 code was added at version 6.11.

temp This optional array is filled with template data. If there is insufficient data to fill it, unused entries are unchanged. An integer or real array can be used. If the template has multiple traces, use temp[points%][traces%] to get real data

and `temp%[points%][traces%]` to get integer data. See `ChanScale()` for an explanation of the use of integer and real data for waveform values.

what%   Omit this or set it to 0 to return the mean template waveform. It is ignored if `n%` is -1. 1 = return the template upper limit, 2 = return the lower limit, 3 = return the template width. The width is half the distance between the upper and lower template limits.

count% This optional integer variable is returned set to the number of events that were accumulated into this template. It is ignored if `n%` is -1.

Returns   If `n%` is positive, the return value is the template code or -1 if the template does not exist. If `n%` is -1, in Edit WaveMark dialogs, the return value is the currently selected sort code of the spike or -1 if you are on-line and the current data did not trigger (background data). In Create WaveMark dialogs the return value is 1 for triggered data and 0 for background data.

See also: `ChanScale()`, `SSButton()`, `SSChan()`, `SSClassify()`, `SSOpen()`, `SSParam()`, `SSRun()`, `SSTempDelete()`, `SSTempInfo()`, `SSTempSet()`, `SSTempSizeGet()`, `SSTempSizeSet()`

SSTempInfo()

This function gets and sets template information for the current channel of the current view, which must be a spike shape window.

```
Func SSTempInfo({n%, item%{, val%{, noLim%}}});
```

n%   If this is omitted, the return value is the number of templates. If present, it is the zero-based index of a template or -1 for all templates (if appropriate). When returning a value, if -1 is used, the value for template 0 is returned.

item%   This selects the information to get or set. Items 0-2 operate on all templates. Items 3-6 operate on visible templates only (`n%` in the range 0 to 19).

      0   Get the index of the first template, starting at `n%`, that has a code of `val%`.
      1   Get the code for template `n%`. It sets the code if `val%` is from 0 to 255.
      2   Get the count of spikes added into template `n%`.
      3   Get the matching event counter. If `val%` is positive, it sets the event count.
      4   Get the lock state of template `n%`. `val%`=0 to clear the lock and >0 to set it.
      5   Change the template width by `val%` pixels. Set `val%` to 1 and -1 to mimic the increase and decrease width buttons. `noLim%` can be used.
      6   Change the template width by val% channel units. `noLim%` can be used.
      7   In collision mode, get or set the exclude checkbox. Set `val%` 1 to exclude, 0 to include. Omit `val%` or set a negative value for no change. You cannot exclude all templates; at least one must be enabled.

val%   Use this argument when setting values and to locate codes when `item%` is 0.

noLim% Set non-zero with item codes 5 and 6 to remove the template width limits that are usually enforced to stop the template becoming too wide or too narrow. Each point of a template has a mean value and a width. It also has a minimum width, which is initialised to the original width when the template was created. The width is allowed to increase to up to 4 times the minimum width, but not to become less that the original. If you allow the width to override these limits, the minimum width is adjusted, as required.

Returns   The value selected by `item%` at the time of the call. The return value is 0 for items 5 and 6. The return value is -1 if a template is not found.

See also: `SSButton()`, `SSChan()`, `SSParam()`, `SSRun()`, `SSTempDelete()`, `SSTempGet()`, `SSTempSet()`, `SSTempSizeGet()`, `SSTempSizeSet()`

SSTempSet()

This function creates a template or adds the current spike in the window to a nominated template in the current channel of the current view, which must be a spike shape view. The first command variant creates a new template from user-supplied data.

The second variant adds the currently displayed data in the window to a nominated template or creates a new template from it. In the Edit WaveMark dialog, this also sets the classification code of the displayed data. If this code is not in the current Marker Filter for the source channel, the displayed data changes to the next available spike. It can also renumber and sort the displayed templates and update online templates in the 1401 to match the templates in the dialog.

```
Func SSTempSet(temp[]{[]}, code%, wide{, count%});
Func SSTempSet({n%{, code%}});
```

temp   Used when creating a new template from arrays of data. This is a one or two dimensional array holding the template shape. An integer or real array can be used. If the template has multiple traces, use `temp[points%][traces%]` for real data and `temp%[points%][traces%]` for integer data. See `ChanScale()` for an explanation of the use of integer and real data for waveform values. If the array is too short or has too few traces, missing values are set to 0.

code%   This is used when creating a new template. Set it to 0 (or omit it in the second command variant) to use the lowest unused code or set it to the code to use for the new template in the range 1-255.

count%   This optional argument has a default value of 1, and sets the number of spikes that contributed to the template.

wide   A number or an array holding the template width, which is half the distance between the upper and lower template limits. If you supply an array, the shape of this array should match the `temp` array.

        If you supply a number, all points are given the same width. This is in user units if `temp` is a real array and is in channel units if `temp` is an integer array.

n%   Optional. If omitted, the currently displayed waveform is added to the best-fit template. Set it to the zero-based index of an existing template to add the current waveform to the template. Set to -1 to add a new template, in which case `code%` determines the code of the new template. Set -2 to renumber the existing templates, in which case `code%` sets the lowest numbered code used. Set -3 to sort the templates into ascending code order. Set -4 to update online templates during sampling.

Returns   The zero-based index of the template that was added or modified or -1 if there was a problem (no matching template, unknown template index or no current waveform). The second variant with `n%` less than -1 returns 0.

See also: `SSTempDelete()`, `SSTempGet()`, `SSTempInfo()`, `SSTempSizeGet()`, `SSTempSizeSet()`

SSTempSizeGet()

This function gets the template size and the displayed data area for the current channel. The current view must be a spike shape window. All arguments are integer variables that are set to the template and display sizes.

```
Func SSTempSizeGet({&start%{, &show%{, &pre%}}});
```

start%   If present, set to the point offset to the template start in the raw data display area.

show%   If present, set to the number of data points in the raw data display area.

pre%   If present, set to the number of pre-trigger points in the raw data display area.

Returns   the number of data points in the template.

See also: `SSTempDelete()`, `SSTempGet()`, `SSTempInfo()`, `SSTempSet()`, `SSTempSizeSet()`

SSTempSizeSet()

This sets the template size and the displayed data area. The template must lie within the displayed data area with at least 2 data points before and after the template. If you set a template area that exceeds these limits, the command attempts to increase the display area (if this is possible). Use YRange() and Optimise() to set the displayed y range,

```
Proc SSTempSizeSet(start%{, n%{, show%{, pre%}}});
```

start%   This is the offset to the beginning of the template from the start of the displayed area. It must be at least 2. Set -1 to leave the start point unchanged.

n%   This sets the number of data points in the template. To keep the current number of points, either omit this argument or set it to -1.

show%   The requested number of points to display in the main data window; it has no effect in the Edit WaveMark dialog. Omit show% or use -1 for no change. If odd, show% is reduced by 1. It is also adjusted if it is too large or too small.

pre%   This sets the number of pre-trigger points to display in the main data window in the range 0 to show%-1. This has no effect in the Edit WaveMark dialog. Omit pre% or set -1 for no change. Out of range values are set to the appropriate limit.

See also:SSClassify(), SSTempDelete(), SSTempGet(), SSTempInfo(), SSTempSet(), SSTempSizeGet(), YRange()

Str$()

This converts a number to a string.

```
Func Str$(x {,width% {,sigd%}});
```

x   A number to be converted.

width% Optional minimum field width. The number is right justified in this width.

sigd%   Optional number of significant figures in the result (default is 6) or set a negative number to set the number of decimal places.

Returns   A string holding a representation of the number.

See also:Asc(), Chr$(), DelStr$(), InStr(), LCase$(), Left$(), Len(), Mid$(), Print$(), Print(), Right$(), Trim(), UCase$(), Val()

Sweeps()

This function returns the number of items accumulated in the result view.

```
Func Sweeps({set%});
```

set%   If present, this sets the number of sweeps held by the view. You might use it to sum waveform averages in a result view so that the sweep count is correct.

Returns   The value returned depends on the type of the result view (see the SetXXXX commands). It is an error to use this in any type of view other than a result view.

See also:SetAverage(), SetEvtCrl(), SetINTH(), SetPhase(), SetPower(), SetPSTH(), SetWaveCrl()

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 Spike2.

```
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, 600=Vista, 601=Windows 7. 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. Add 1000 to
       get the pixel co-ordinates that exclude the system taskbar or any other reserved
       screen area.

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()

System$()

This returns the operating system name and accesses Spike2 environment variables. The
environment holds a list of strings of the form "name=value". You can get or set the
value associated with name. You can also read all the 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.

Each process has its own environment. A process started with ProgRun() inherits a copy
of the Spike2 environment, so you can pass it information. However, you cannot see
environment changes made by the new process. Here are some examples of use:

```
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;
```

See also:ProgRun(), System()

TabSettings()

This sets and gets the tab settings for a text view. Any changes you make apply to the
current view only. If you want to change the tab settings for all views, open the Edit
menu preferences General tab and click the appropriate button in the Text view settings.
It is possible to do this using a script with the Profile() command.

```
Func TabSettings({size%{, flags%}});
```

size% Tab sizes are set in units of the width of the space character in the Default
       style set for the view (style 32). Values in the range 1 to 100 set the tab size. If
       size% is 0 or omitted, no change is made. If size% is -1, the return value is the
       current flags% value for the text view.

flags% If omitted, no change is made to the flags. Otherwise, this is the sum of flag
       values: 1=Keep tab characters (the alternative is replace tabs with spaces),
       2=show indents.

Returns If size% is positive, the return value is the tab size at the time of the call. If
        size% is -1, the return value is the flags% value at the time of the call.

See also:FontGet(), FontSet(), Profile()

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°. π is approximately 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()`, `Ln()`, `Log()`, `Pow()`, `Sin()`, `Sqrt()`

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()`

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 `TimeDate()`. To obtain relative time and fractions of a second, use `Seconds()`.

```
Func Time$({tBase%, {show%, {amPm%, {sep$}}}});
```

tBase% Specifies the time base to show the time in. You can choose between 24 hour or 12 hour clock mode. If this argument is omitted, 0 is used.

    0 Operating system settings   2 12 hour format
    1   24 hour format

show% Specifies the time fields to show. Add the values of the required options together and use that 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   4   Show seconds
    2   Show minutes   8   Remove leading zeros from hours

amPm% This sets the position of the "AM" or "PM" string in 12 hour format and has no effect in 24 hour format. If omitted, a value of zero is used. The string that gets printed ("AM" or "PM") is specified by the operating system.

    0   Operating system settings   2   Show to the left of the time
    1   Show to the right 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$()`, `FileTime$()`, `Seconds()`, `TimeDate()`

TimeDate()

This returns the time and date in seconds, minutes, hours, days, months, and years plus the day of the week. You can use separate variables for each field or an integer array. To get the data or time as a string, use `Date$()` or `Time$()`. To measure relative times, or times to a fraction of a second, see the `Seconds()` command. To get the current sampling time, see `MaxTime()`.

```
Proc TimeDate(&s%, {&m%, {&h%, {&d%, {&mon%, {&y%, {&wDay%}}}}}});
Proc TimeDate(td%[])
```

s%   If s% is the only argument, it is set to the number of seconds since midnight. Otherwise, it is set to the seconds since the start of the current minute.

m%   If this is the last argument, it is set to the number of minutes since midnight. Otherwise, it is set to the number of full minutes since the start of the hour.

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 2002.

wDay%   If present, the day of the week will be returned here as 0=Monday to 6=Sunday.

td%[]   If an array is the first and only argument, the first seven elements are filled with time and date data. 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$(), MaxTime(), Seconds(), Time$()

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. 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).

You can define up to 40 visible buttons, numbered from 1, in your toolbar. There is an invisible button 0, which sets a function that is called while the toolbar waits for a button to be pressed. Buttons can be linked to the keyboard. However, any keys linked to buttons belong to the toolbar when it is active and waiting for a button press. If you link the A key to a button, each time you press A, the button is pressed, even if the text caret is in a text window. You can also link mouse movement and button presses to script functions and control the mouse pointer.
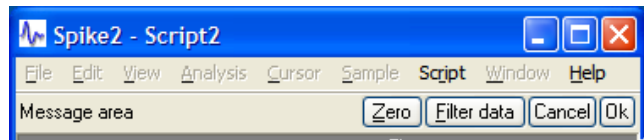
See also:Interact(), Toolbar(), ToolbarClear(), ToolbarEnable(), ToolbarMouse(), ToolbarSet(), ToolbarText(), ToolbarVisible()

Toolbar()   This function displays the toolbar and waits for the user to click a button or press a linked key. If button 0 is 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 state is illegal and an error is returned. If the toolbar was not visible, it becomes visible when this command is given.

When the Toolbar is active, Spike2 ignores the "escape" key (command-dot on the Macintosh, Esc on the PC) unless an "escape button" has been set by ToolbarSet().

```
Func Toolbar(text$, allow% {,help%|help$});
```

text$ A message to display in the message area of the toolbar. The message area competes with the button area. With many buttons, the text may not be visible.

allow% A code that defines what the user can do (apart from pressing toolbar buttons). See Interact() for the allowed values.

help   This is either the number of a help item (CED internal use) or it is a help context string. This is used to set the help information that is presented when the user

presses the F1 key. Set 0 to accept the default help. Set a string as displayed in the Help Index to select a help topic, for example "Cursors: Adding".

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 or mouse up function. If the toolbar closes due to a ToolbarMouse() mouse up function returning 0, the return value is positive and greater than any toolbar button.

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 removes some or all of the toolbar buttons. If you delete all buttons, the Toolbar() function inserts an OK button, 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 button to clear. Buttons are numbered from 0. If omitted, all buttons are cleared. If the toolbar is visible, changes are shown immediately.

See also: Interact(), Toolbar(), ToolbarSet(), ToolbarText()

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. 0 disables a button, 1 enables it.

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()

ToolbarMouse()

This command gives you access to the mouse positions and left button mouse clicks in Time, Result and XY views when the mouse is over a data channel while a toolbar is active. There is an example, here.

```
Proc ToolbarMouse(vh%, ch%, mask%, want%, Down%{, Up%{, Move%}});
```

vh% Either the view handle of the view that you want to get mouse information from, or 0, meaning that you will accept mouse information from any suitable view.

ch% Either a channel number in the view that you want mouse information for when you are over it or 0 to accept input from any channel. In an XY view, the display area is treated as belonging to channel 1, so setting 1 or 0 will work. If you set a channel number, once you have clicked on that channel, all values passed to you will be for that channel, even if you drag the mouse over a different channel. To click on a channel, then drag to another and be told about the other channel you

must set `ch%` to 0. With `ch%` set to 0, if you drag to a place where there is no channel, you will be returned the last position that was over a channel.

`mask%` This, and the next argument (`want%`) are used when the left mouse button is clicked to decide if the script should be told about the mouse click. When the mouse button is clicked, and the conditions set here are met, the mouse becomes owned by the script and all mouse input will be given to the script until the mouse is released (or another application grabs the mouse). The mouse is said to be *captured*. The conditions set here are also used to decide if the script should be informed of mouse movements when the mouse is not owned by the script. Both `mask%` and `want%` are the sum of a set of values:

1   The left-hand mouse button is down.
2   The right-hand mouse button is down (releasing this button will normally display a context menu)
4   The `Shift` key is down
8   The `Ctrl` key is down
16   The mouse middle button is down
32   Extra button 1 is down (this is the left-hand side button on my mouse)
64   Extra button 2 is down (this is the right-hand side button on my mouse)
128   The `Alt` key is down
256   This mouse down was a double-click

The `mask%` value determines which of these items we care about. For example if you cared about the state of `Shift` and `Ctrl`, you would set the value to 12.

`want%` This argument sets the desired state of the items that you have identified with `mask%`. For example, if you only want to be told when the `Shift` key is down and the `Ctrl` key is not down, set `mask%` to 12 and `want%` to 4. Another use would be to stop the script being told when the mouse was just being moved around but had not been clicked in an area we wanted. In this case you would set `mask%` and `want%` to 1 (only tell me when the left-hand mouse button is down).

`Down%` This is the name of a user-defined function that is called when the mouse left-hand button is clicked and the conditions implied by `vh%`, `ch%`, `mask%` and `want%` are satisfied. The arguments and return value are described below.

`Up%` This is the name of an optional user-defined function that is called when the mouse button is released after it has been captured. You will always get a `Down%` function call before you get an `Up%` call. If another application (rudely) takes over the mouse by popping up a window, you will also get a call to the `Up%` function. This function is described below.

`Move%` This is the name of an optional user-defined function that is called when the mouse is moved after being captured. This function is also called when the mouse is moved when not captured by a mouse click and the conditions set by `vh%`, `ch%`, `mask%` and `want%` are satisfied. If you only want to be called during a drag operation, make sure you include the value for the left-hand mouse button in both `mask%` and `want%`

The user-defined mouse functions All three functions have exactly the same arguments. The function names do not have to be `Down%`, `Up%` and `Move%`, you can choose any suitable names. Ideally your mouse functions (especially mouse move) should not take a long time to run; if they do, the mouse movement will feel uncomfortable and jerky. The mouse up function could be used to trigger a more time consuming operation. The return value has different uses in all three cases. The functions are:

```
Func Down%(vh%, ch%, x, y, flags%);
Func Move%(vh%, ch%, x, y, flags%);
Func Up%(vh%, ch%, x, y, flags%);
```

`vh%` The view handle of the view that the mouse is over. If you set `vh%` to a view handle value in the `ToolbarMouse()` call, then this will be that value.

ch%   The channel number that the values of x and y relate to. If you specified a channel number in the ToolbarMouse() call, then this will be that value.

x   The x-axis value in x-axis units. If you click and drag you can get values that are outside the visible range of the x axis. It is OK to scroll the view in response.

y   The y-axis value in y axis units for the channel identified by ch%. If there is no y axis, the value will be 0.

flags% This holds the same information as held by mask% and want%. It gives you the state of the mouse buttons and Shift, Ctrl and Alt keys.

Return value   The return values have different uses for the three functions:

Func Down%()   If the click was not over anything interesting or you decide that you do not want to handle it, then return 0 and Spike2 will decide what to do with the click. You will not get a mouse down call when the mouse is over the XY view key, or over a vertical or horizontal cursor. However, you do get priority over Spike2 for all other clicks in a view (for sizing, for instance). Return values greater that 0 select the mouse pointer to display (this is covered below) and mean that you want to capture the mouse for script use.

Add one of the values below to the return value to display an indication of a selection size or area. If you add any of the values and drag beyond the left or right edges of the data area, the area will scroll (if it is allowed to).

Value   Result

256   Display a selection rectangle, as you would for zooming in and out. If you return 256, the appropriate zoom cursor is selected based on the state of the Ctrl key).

512   Display a measurement of the distance between the start of the drag and the current position. If you return 512, the measurement cursor is selected.

1024   Display a line from the selection start to the current position. If you return 1024, the measurement cursor is selected.

Func Move%()   The return value from this sets the mouse pointer to be used (see below). A return value of 0 makes no change to the mouse pointer, which means that the pointer set by Func Down%() will be used for a drag operation and the cross-hair cursor (#1 in the list below) used when not dragging.

Func Up%()   The return value from this determines if the toolbar closes or not. Return 0 or a negative number to close the toolbar, 1 to keep running.

Mouse pointers set by   You have access to many of the mouse pointers that are available in Spike2, so you can
return values   use these to indicate that you are over an item or to show that you are dragging something. The values for the preset mouse pointers are:

Codes 1-7 and 23-26 are system mouse pointers and may have a different appearance if you have chosen a custom set of mouse pointers in the Windows system Control Panel in the Mouse section. 3D and animated mouse pointers can be a lot slower than simple monochrome pointers on some systems. You can also define your own mouse pointers with the MousePointer() command. These are assigned numbers above the range of the built-in cursors.
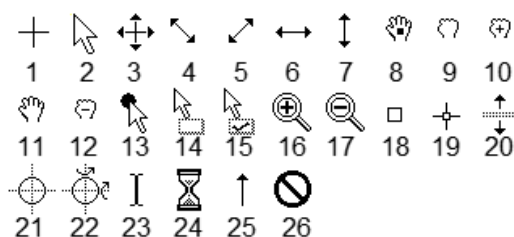
Mouse double-click   If you double-click, you will get a mouse down, possibly followed by one or more mouse
behaviour   moves, followed by a mouse up for the first click. The second click generates a mouse down with the double-click flag set in the flags. If your mouse down function does not

claim the mouse (returns 0), Spike2 will use the double click as appropriate for the channel in which you clicked.

See also:`MousePointer(), Toolbar(), ToolbarClear(), ToolbarEnable(), 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. There are two variants:

```
Func ToolbarSet(item%, label$ {,func ff%()});          'Set a button
Func ToolbarSet();         'Get the last used button number while the toolbar is active
```

item%   The button number in the range 1 to 40 to add or replace or 0 to set or clear a function that is called repeatedly while the toolbar waits for a button press.

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"`.

To link a key to a button, place `&` before a character in the label or add 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 label `"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 Enter
0x6d nk -	0x6e nk .	0x6f nk /	0x70-0x87 F1-F24

Use of other keys 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 with no arguments. 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 runs. If the `item%` 0 function is set, that 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.

Returns 0 unless called with no arguments when it returns the last used button number.

See also:`Asc(), Interact(), Toolbar(), ToolbarClear(), ToolbarEnable(), ToolbarText(), ToolbarVisible()`

ToolbarText()

This replaces any message in the toolbar, and makes the toolbar visible if it is invisible. It 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 this is 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()

Trim(), TrimLeft(), TrimRight()

These functions remove leading or trailing whitespace (spaces, tabs and end of line characters) or user-defined characters from a string variable. TrimLeft() removes leading characters, TrimRight() removes trailing characters and Trim() remove both.

```
Proc Trim(&text${, chars$});
Proc TrimLeft(&text${, chars$});
Proc TrimRight(&text${, chars$});
```

text$ The string variable to remove characters from.

chars$ An optional list of characters to remove. If omitted, " \t\n\r" is used.

See also:DelStr$(), InStr(), Left$(), Len(), Mid$(), Right$()

Trunc()

Removes the fractional part of a real number or array. To truncate a real number to an integer, assign the real to the integer. ArrConst() copies a real array to an integer array.

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

x   A real number or a real array.

Returns 0 or a negative error code for an array. For a number it returns the value with the fractional part removed. Trunc(4.7) is 4.0; Trunc(-4.7) is -4.0.

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

U1401 commands

The U1401 commands give you access to the CED 1401 interface connected to your computer. You cannot use a 1401 that Spike2 is sampling from. Use U1401Open() to take control of the 1401, and U1401Close() to release it. See the *1401 family programming manual* for details of using the 1401; this is available from CED as part of the 1401 programming kit and as a download from the CED web site.

See also:U1401Close(), U1401Ld(), U1401Open(), U1401Read(), U1401To1401(), U1401ToHost(), U1401Write()

U1401Close()

This command closes the link between the script language and a 1401 interface generated by the U1401Open() command. If there is no open 1401, the command is ignored.

```
Proc U1401Close();
```

See also:U1401Ld(),U1401Open(),U1401Read(),U1401To1401(),
U1401ToHost(),U1401Write()

U1401Ld()

This command loads one or more 1401 commands with the option of nominating the folder to load the commands from. If no 1401 is open, the script halts.

```
Func U1401Ld(list${, path$});
```

list$ The list of commands to load separated by commas. Include KILL as the first item to clear all commands first. For example: "KILL,ADCMEM,MEMDAC". The file name for a command is the command name plus an extension that depends on the type of 1401. The extension is added automatically.

path$ Optional. The path to the folder to search for the commands. If omitted, or if the command is not found in this path, the 1401 folder in the Spike2 source folder is searched, then any path indicated by the 1401DIR environment variable, and finally, the \1401 folder on the current drive.

Returns 0 if all commands loaded. Otherwise, the bottom 16 bits is an error code and the upper 16 bits is the 1-based index to the command in the list that failed to load.

See also:U1401Close(),U1401Open(),U1401Read(),U1401To1401(),
U1401ToHost(),U1401Write()

U1401Open()

This command attempts to open a 1401 for use by other U1401 commands and returns the type of the opened 1401 or a negative error code. It is not an error to call U1401Open() multiple times with no intervening U1401Close().

```
Func U1401Open({unit%});
```

unit%   Optional 1401 unit number (taken as 0 if omitted) in the range 1 to 8 or 0 for the first available unit. You cannot open a 1401 that is in use for data sampling; you cannot sample data with a 1401 that is held open by U1401Open().

Returns The return value is the type of the 1401 detected: 0=standard 1401, 1=1401plus, 2=micro1401, 3=Power1401, 4=Micro1401 mk II, 5=Power1401 mk II, 6=Micro1401-3, 7=Power1401-3. Otherwise, it is a negative error code that can be decoded by Error$().

See also:Error$(),U1401Close(),U1401Ld(),U1401Read(),U1401To1401(),
U1401ToHost(),U1401Write()

U1401Read()

This command reads a text response from a 1401 and optionally converts it into one or more integer values or reports the number of available input lines. If no 1401 is open, the script halts. There are four variants:

```
Func U1401Read();                            Get count of input lines
Func U1401Read(&text$);                       Read an input line as text
Func U1401Read(&v1%{, &v2%, {...}});     Read a line and convert to integers
Func U1401Read(arr%[]);              Read a line, convert to integer array
```

text$ A text variable returned holding the entire response.

v1%   An integer variable that is returned with the first integer number read.

vn%   Optional integer variables (v2% up to v12%) returned with following values. Values for which no number is returned are unchanged.

arr%[] An integer array that is filled (starting at element 0) with converted values.

Returns The version with no arguments returns the number of available input lines. The other versions return the number of items that were converted from the input text and stored to a script variable or a negative error code. If you use a variant that reads a line and there is no text to read, the command times out after about 3 seconds and returns an error code.

See also:U1401Write()

U1401To1401()

This command transfers the contents of an integer array to memory in the 1401.

`Func U1401To1401(arr%[]{[]}, addr%{, size%});`

arr%   This is a one or 2 dimensional array to transfer. If you use a 2 dimensional array to interleave 4 channels of data, for example for MEMDAC, set the first dimension to 4 and the second to the number of points per channel.

addr%   The start address of the block of contiguous memory in the 1401 user area to be filled with data.

size%   Optional, the number of bytes in the 1401 that each array element is copied to. Acceptable values are 1, 2 or 4. If size% is omitted, 4 is used.

Returns 0 if the data was transferred without a problem, or a negative error code.

See also:U1401Close(),U1401Ld(),U1401Open(),U1401ToHost()

U1401ToHost()

This command transfers a block of 1401 memory into an integer array.

`Func U1401ToHost(arr%[]{[]}, addr%{, size%});`

arr%   This is a one or 2 dimensional array to receive the data. If you use a 2 dimensional array to interleave 8 channels of data, for example for ADCMEM, set the first dimension to 8 and the second to the number of points per channel.

addr%   The start address of the block of contiguous memory in the 1401 user area to copy data from.

size%   Optional, taken as 4 if omitted. The number of bytes of 1401 data used to set each array element. Use 1, 2 or 4 to read 1, 2 or 4 bytes and sign extend to 32-bit integer. Use -1, -2 or -4 to read 1 or 2 or 4 bytes and zero extend to 32-bit integer.

Returns 0 if the data was transferred without a problem, or a negative error code.

See also:U1401Close(),U1401Ld(),U1401Open(),U1401Read(),U1401To1401()

U1401Write()

This command writes a text string to the 1401.

`Func U1401Write(text$);`

text$ The text to write to the 1401. Commands to the 1401 are terminated by either a newline "\n" or a semicolon ";".

Returns 0 if the line was added to the 1401 output buffer, or a negative error code.

See also:U1401Close(),U1401Open(),U1401Read(),

UCase$()

This function converts a string into upper case. The upper-case operation may be system dependent. Some systems may provide localised uppercasing, 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 case version of the original string.

See also: `Asc()`, `Chr$()`, `DelStr$()`, `InStr()`, `LCase$()`, `Left$()`, `Len()`, `Mid$()`,
         `Print$()`, `ReadStr()`, `Right$()`, `Str$()`, `Trim()`, `Val()`

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%{, flag%}});
```

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.

flag%   If present and 1, also accept hexadecimal as: `{white space}0x|Xhexdigits`

Returns   It returns the extracted number, or zero if no number was present.

See also: `Asc()`, `Print$()`, `ReadStr()`, `Right$()`, `Str$()`, `UCase$()`

VerticalMark()

This sets up, cancels or gets information about vertical markers in the current time view. The View menu Vertical Marker command documentation has more information. Use the `FontGet()` and `FontSet()` commands to set the text font.

```
Func VerticalMark(chan%{, flags%{, just%{, hPos%{, dir%}}}});
```

chan%   Either the channel number of an event, Marker or extended marker type, 0 to cancel any vertical marker or -1 to -5 to return the current values of arguments 1 (channel) to 5 (direction).

flags%   This is the sum of: 1=Draw on top of channel data, 2=Use Marker code colours, 4=Enable text for TextMark channels, 8=No vertical line for text, 16=No fill behind text. Use -1 for no change.

just%   Sets the text justification as: 0=Left, 1=Centre, 2=Right. Use -1 for no change.

hPos%   Sets the text baseline: 0=Below, 1=On, 2=Above the line. Use -1 for no change.

dir%   Sets the text direction: 0=Upwards, 1=Downwards. Use -1 for no change.

Returns   Calls with chan% greater than or equal to 0 return 0 for success or a negative error code. Calls with chan% less than 0 return the requested information.

See also: `FontGet()`, `FontSet()`

View(), View().x() and View().[ ]

The `View()` function sets the current view and returns the last view handle. 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 of a view that is to be made the current view. Use `ViewKind()` to test for a valid handle.
       0   (or omitted) no change of the current view is required.

<0   If `vh%` is `-n`, this selects the nth duplicate of the time view that is associated
     with the current view. The current view can be a time, result or XY view. If
     the current view is a time view, this is equivalent to `Dup(n).View(-1)` in
     a result or XY view returns the time view from which it was created.

Returns   The handle of the view that was current at the time of the call. If an argument is
          passed in which is not valid, the script stops with an error.

View().x()   The `View().x()` construction overrides the current view for the evaluation of `x()`. For
     example, `View(vh%).Draw(1,2)` draws view number `vh%`. It is an error if the selected
     view does not exist or if the function closes the original view, and the script stops.

```
View(vh%).x()
```

`vh%`   A view handle of an existing view, 0 for the current view, or -n for the nth
        duplicate of the time view associated with the current view.

The equivalent code to `View(vh%).x()` is:

```
var temp%;
temp% := View(vh%);  'Save the current view
x();                 'call the user-defined or built-in function
View(temp%);         'restore the original view
```

This means that `View(vh%).FileClose()` causes an error if `vh%` is the current view.

View().[]   The `View().[]` construction overrides the current view to give you access to the data
     arrays that form result view channels.

```
View(vh%{,ch%}).[]
```

`vh%`   A view handle of an existing result view or 0 for the current result view.

`ch%`   An optional result view channel number. If omitted, 1 is assumed.

For example:

```
ArrConst(View(0,2).[6:20], 0); 'zero 20 elements of channel 2
View(rv%).[4] := 3;            'set fifth element of channel 1
```

See also:`App()`,`Dup()`,`FileClose()`,`FrontView()`,`SampleHandle()`,
`ViewFind()`,`ViewKind()`,`ViewList()`

ViewColour()

This function gets and sets the colours of time, result and XY view items, overriding the
application-wide colours set by `Colour()`. Use `ViewColourGet/Set()` in new scripts.

```
Func ViewColour(item%{, col%});
```

`item%`   The colour item to get or set; 0=background. There is only one item at present.

`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()`

ViewColourGet()

This function gets the RGB colour of time, result and XY view items, indicating if it
overrides the application-wide colours set by `ColourSet()`.

```
Func ViewColourGet(item%{, &r, &g, &b});
```

`item%`   The colour item to get or set; 0=background. There is only one item at present.

`r g b`   If present, set to the red, green and blue colour values in the range 0.0 to 1.0.

Returns   1 if the returned colour overrides the application colour, 0 if it does not.

See also:ChanColourGet(), ColourGet(), ViewColourSet()

ViewColourSet()

This function sets the RGB colour of time, result and XY view items, to override the application-wide colours set by ColourSet().

```
Proc ViewColourSet(item%{, r, g, b});
```

item%   The colour item to get or set; 0=background. There is only one item at present.

r g b   If present, These set the red, green and blue colour values in the range 0.0 to 1.0. If omitted, the item colour is set to the application default.

See also:ChanColourSet(), ColourSet(), ViewColourGet()

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.

Returns The view handle of a view with a matching title, or 0 if no view matches.

See also:WindowTitle$(), View(), ViewList()

ViewKind()

This function returns the type of the current view or a view identified by a view handle. Types 5-7 are reserved. Type 11 windows include the sampling configuration window and control panel, the sequencer control panel and multimedia windows.

```
Func ViewKind({vh%});
```

vh%   An optional view handle. If omitted, the type of the current view is returned. Use -n in a result or XY view to find the nth duplicate of the time view from which the current view was created.

Returns The type of the current view or the view identified by vh%. View types are:

-2 Invalid handle	1 Text view	4 Result view	10 Application window
-1 Unknown type	2 Output sequence	8 External text file	11 Other types
0 Time view	3 Script	9 External binary file	12 XY view

See also:ChanKind(), SampleHandle(), ViewList()

ViewLineNumbers()

This function is used in a text-based view to display or hide line numbers and to set the number of decimal digits of space to use for a line number.

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

show%   Set -1 or omit for no change, 0 to hide line numbers, 1 to show them. Use 2-8 to set 2-8 digits of space and greater than 8 for a standard display (5 digits).

Returns 0 if line numbers were hidden at the time of the call, else the display space (2-8).

See also:Gutter()

ViewLink()

This function returns the view handle of the view that owns the current window. For example, you can use this to get the time view that owns a multimedia, spike shape or spike monitor window or the time view that created a result or XY view. This is slightly

different from `View(-n)`, which finds the nth duplicate of the time view linked to the current time, result or XY view.

```
Func ViewLink();
```

Returns The handle of the linked view, or 0 if there is no such view.

It can also iterate through process operations that are linked to the current time view. For example, if you create a new sampling document using `FileNew()` with `mode%` containing 2, you may also open other result and XY documents and have additional channels present due to the sampling configuration. In this case, the command is:

```
Func ViewLink(n%{, mask%{, &name$}});
```

n%   This can be 0, meaning count the number of processes and return it, or it can be the process number to report on.

mask%   This determines the types of processes we wish to count or report on and is the sum of: 1 for result views generated by `SetINTH()` and similar calls, 2 for XY views generated by `MeasureToXY()` and 4 for data channels in the current view generated by `MeasureToChan()`. If this argument is omitted, it takes the value 3, to report on result and XY views. Normally this argument will be 1, 2, 3 or 4. If you use 5, 6 or 7 with n% > 0 the return value may be a channel or a view.

name$ If present, and n% is greater than 0, this is set to the name of the command that will process the data.

Returns If n% is 0, this returns the number of processes that match the mask% argument. If n% is greater than 0, then this returns either the handle of the view or the channel number in the current view that is the target of the process operation. If n% is greater than 0 and there is no corresponding process, the return value is 0.

Note that this form of the command identifies active processes, that is processes for which the `Process()` command could have some effect.

See also:`App()`, `SampleHandle()`, `View()`, `ViewKind()`, `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 (element 0) is filled with the number of handles returned. If the array is too small to hold the full number, the numbers that will fit are returned.

types% The types of view to include. This code 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 Time views	8 Script views	512 External binary	4096 XY views
2 Text views	16 Result 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 number of windows that match `types%`.

The following example prints all the window titles into the log view:

```
var list%[100],i%;       'Assume no more than 99 views
ViewList(list%[]);       'Get a list of all the views
for i%:=1 to list%[0] do 'Iterate round all the views
```

```
    PrintLog(view(list%[i%]).WindowTitle$()+"\n");
    next;
```

See also:`App()`, `SampleHandle()`, `ViewKind()`

ViewMaxLines()

This function gets and optionally sets the maximum number of lines that the current text view will retain if text is added by a script. For the Log view, this value is initially set by the Edit menu Preferences in the General tab, but a script can set or clear the line limit for any text-based view. `ViewStandard()` clears the limit except in a Log view, where it sets the value set in the preferences.

```
Func ViewMaxLines({max%});
```

max%   Set -1 or omit for no change, 0 for no limit or the maximum number to display.

Returns The maximum number of lines at the time of the call.

Each time the line limit is exceeded, the number of lines in the file is reduced to 90% of the maximum line count by deleting the lowest numbered lines. We reduce the line count to 90% of the maximum because deleting lines from the start of the view is a slow operation (in very large views); by allowing the view to grow normally for a while this minimises the time penalty for setting this option. For example, if max% is set to 1000, the next time that a line is added and more than 1000 lines are in the file, the line count is reduced to 900. After this 100 lines can be added before the limit is tripped.

See also:`ViewStandard()`

ViewOverdraw()

All time views hold a list of overdraw trigger times sorted into ascending order. The use of this list is enabled by `ViewTrigger()`. This command adds new times to the list if overdraw mode is enabled. The command can also read back the list of trigger times. Whenever this command changes the list, the view is positioned to the last list time. It is the script equivalent of the View menu Overdraw dialog. Trigger times are taken from a data channel, or can be supplied as times. There are two command variants:

```
Func ViewOverdraw(flag%, chan%, sTime, eTime);
Func ViewOverdraw({flag%{, time|times[]}});
```

flag%   If -1, the times[] array must be present and times are read back from the list. Otherwise, it is the sum of the following values:

1   Empty the list before optionally adding any new values. If omitted, previous events are preserved unless adding new events causes them to be deleted.
2   Adding a time does not delete all later times in the list. If omitted, adding a time deletes all later times; the last time added becomes the current time.

chan%   The channel number of an event, Marker, or Marker-derived channel in the time view to use as a source of trigger times. `ViewTrigger(-1)` returns the channel that the times in the list come from or -1 if there is a mix of channels.

sTime   The start of the time range to search for times.

eTime   The end of the time range to search for times.

time   A single time to be added to the list.

times   An array of times to be added to the list (flag% >= 0) or to hold times copied from the list (flag%=-1). Times are added in array order, so the times should be in ascending order if flag% is not 2 or 3. `ViewTrigger(-1)` will return -1.

Returns If flag% is -1, the return value is the number of elements of times[] that were filled with times. Otherwise, it is the resulting number of times in the list or -1 if overdrawing is not enabled.

See also:`ViewTrigger()`, `ViewOverdraw3D()`

ViewOverdraw3D()

This command controls 3D drawing (enabled by the `ViewTrigger()` command or the Display Trigger dialog). It is the equivalent of the View menu Overdraw 3D dialog. This command may only be used in a Time view. There are two command variants. The first sets the overdraw values, the second reads back the current settings.

```
Func ViewOverdraw3D(xProp, yProp{, xScale{, yScale{, flags%}}});
Func ViewOverdraw3D(get%)
```

xProp   The proportion of the available x space (in the range 0 to 1.0) to use for the 3D effect. Values outside the range 0 to 1 are limited to this range.

yProp   The proportion (range 0 to 1) of the available y space to use for the 3D effect.

xScale This optional argument sets how much to shrink the display width when going from the front to the back to give a perspective effect. Values are limited to the range 0 (shrink to nothing) to 1 (no shrink). If omitted, no change is made.

yScale How much to shrink the display height when going from the front to the back to give a perspective effect, in the range 0 to 1. If omitted, there is no change.

flags% This optional argument is omitted, no change is made. It the sum of:

1   Display a fixed count of frames or time range. This only has any effect if a maximum frame count or time range is set by `ViewTrigger()`. If not set, the latest frame is at the front and the oldest is at the back.

2   The position of a frame is determined by the frame time. If omitted, the position of a frame is set by the frame number.

get%   The variant of the command with one argument uses this value to indicate the value to read back: -1=xProp, -2=yProp, -3=xScale, -4=yScale, -5=flags%

Returns 0 when setting a value or the value requested by `get%`.

See also:`ViewTrigger()`, `ViewOverdraw()`

ViewStandard()

This sets the current time, result or XY view to a standard state by making all channels and axes visible in their standard drawing mode, axis range and colour. All channels are given standard spacing and are ungrouped and the channels are sorted into the numerical order set by the Edit menu Preferences. In a time view, duplicate channels are deleted, triggered mode is disabled and any channel processing is removed. In an XY view the key is hidden and the axes are optimised. In a text-based view is restores the view to the default set of styles, removes any text zoom and displays the standard margins.

```
Proc ViewStandard();
```

See also:`ChanOrder()`, `ChanWeight()`, `DrawMode()`, `XYDrawMode()`

ViewTrigger()

This controls the triggered drawing mode available for time views; it is a fatal error to use this command in a different view type. This command is the equivalent of the View menu Display Trigger command. See that for a detailed description of the arguments.

```
Func ViewTrigger(chan%, pre, hold, xZero%{, cur0%{, wait{,
                    over%{, col%{, maxO%{, maxT}}}}}}); Set and enable
Func ViewTrigger({mode%});              Enable/disable, step and get information
```

chan%   The event, marker, WaveMark, TextMark or RealMark trigger channel. Set 0 for a paged display on-line or during rerun.

pre   The pre-trigger time to display, in seconds.

hold   The minimum hold time for on-line displays, the minimum time to the next/previous trigger event for off-line use.

xZero% If this is set to 1, the x axis zero point (for display only) is moved to the trigger time. All time measurements are still relative to the start of the file.

cur0% This sets the cursor 0 action on each view trigger. 0=no action, 1=move cursor 0 without causing the active cursors to iterate, 2=move cursor 0 and update any active cursor positions after the time period set by wait. If omitted, 0 is used.

wait   If cur0% is 2, this sets how long to wait online before updating the active cursors. If omitted, 0 is set. Set -1 to leave the current wait value unchanged.

over% Set 1 to enable overdraw, add 2 to enable 3D drawing, 0 to disable, -1 for no change. If omitted, 0 is used.

col%   Overdrawn data colour. -1 or omitted=no change, 0=normal, 1=half intensity, 2=fade to background, 3=fade to secondary.

maxO% Maximum overdraws from 1 to 4000, 0 for no limit, -1 or omit for no change.

maxT   Maximum time range of overdrawn traces up to 10000 seconds or 0 for no limit or -1or omitted for no change.

mode% This command version gets information, moves to the next and previous trigger, and enables and disables the trigger. With no arguments, the command returns the enabled or disabled state as 1 or 0. If there is a single argument, it can be:

    3,2 Move to the next (3) or previous(2) trigger and return the trigger time or -1 if no trigger is found or if the trigger is not enabled.

    1 Enable the View trigger with the current settings, return previous state.

    0 Disable the trigger and return the previous trigger state.

    Negative values return the current argument values:

| -1 | chan% | -3 | hold | -5 | cur0% | -7 | over% | -9 | maxO% |
| -2 | pre | -4 | zZero% | -6 | wait | -8 | col% | -10 | maxT |

Returns When the command is used with mode%, the command returns the requested information or -1 if the information is not available. All other use returns the enabled/disabled state as 1 or 0, or a negative error code.

See also:DrawMode(), ViewOverdraw(), ViewOverdraw3D(), ViewStandard()

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 Spike2 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:ChanColour(), Colour(), XYColour()

ViewZoom()

This function is used in a text-based view to get and optionally set the zoom factor. This is the number of points to add to the nominal size of the text.

```
Func ViewZoom({zoom%});
```

zoom%   Omit for no change or a value in the range -10 to 20 to add to the point size of all the text. The resulting minimum text size is 2 point regardless of zoom%.

Returns The zoom value at the time of the call.

See also:ViewStandard()

VirtualChan()

This command controls virtual channels in the current time view and has the functionality of the Virtual Channel dialog. Virtual channels are defined by an algebraic expression that can involve other channels (but not virtual ones). The first command variant creates and modifies a virtual channel; the second reports the state of a virtual channel.

```
Func VirtualChan(chan%, expr${, match%{, binsz{, align}}});
Func VirtualChan(chan%, get%{, &expr$});
```

chan%   In the first command variant, 0 creates a new virtual channel and returns the channel number. The remaining arguments set initial channel values. If not 0, this is the number of an existing virtual channel to modify or from which to read back the settings.

expr$ In the first command variant, this is a string expression defining the output. See the Analysis chapter of the Spike2 manual for details. In the second command variant, this optional string variable returns the current channel expression.

match%   This is the number of an existing waveform, RealWave or WaveMark channel to match for sample interval and alignment. If 0, the sample interval and alignment are set by the binsz and align arguments. If negative, no change is made.

binsz   If match% is 0, this sets the sample interval of the channel. Values of binsz less than the file time resolution are ignored, as are values less than or equal to 0.

align   If match% is 0, this sets the channel alignment. Values less than 0 are ignored.

get%   In the second command variant, get% determines the returned value. 0 = the state of the expression parsing, 1 = match%, 2 = binsz, 3 = align.

Returns When creating a channel, it returns the new channel number or a negative error code. When modifying an existing channel, it returns 0 for OK, a negative code if the match%, binsz or align arguments are illegal, or a positive code if the expression contains an error. The second variant returns the get% information or a negative error code. If get% is 0, the return value is negative if the channel is not a virtual channel, 0 if the expression is acceptable and a positive code if not.

See also:MemChan()

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.

```
Func Window(xLow, yLow{, xHigh{, yHigh{, scr%{, rel%}}}});
```

xLow   Position of the left hand edge of the window in percent. 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 in percent.

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 or -1, positions are relative to the application window. Otherwise, 0 selects the entire desktop rectangle and greater values select a particular screen rectangle (but see rel%). See System() for more screen information.

rel%   Ignored unless scr% is 0 or greater. Set 0 or omit for positions relative to the intersection of scr% rectangle and the application window, 1 for positions relative to the scr% rectangle. When positioning the application window, rel% is treated as if it were 1.

Returns 1 if the position was valid, or -1 if the rectangle set by scr% and rel% is of zero size. Before Spike2 version 6.06 this was a Proc with no return value.

Examples
```
View(App()).Window(0,0,100,100,0);  'Spike2 uses all desktop...
View(App()).Window(0,0,100,100,2);  '...uses all second monitor
Window(0,0,100,100); 'current view uses all application space
```

See also:App(), System(), WindowDuplicate(), WindowGetPos(), WindowSize()

WindowDuplicate()

This duplicates the current time window, creating a new window that has all the settings of the current window. 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 The view handle of the new window, a negative error code or 0 if there are no free duplicates. There is a limit of 64 duplicates per window.

See also:Window(), WindowGetPos(), WindowTitle$(), WindowVisible()

WindowGetPos()

This gets the window position of the current view with respect to the application window. Positions are measured from the top left-hand corner as a percentage.

```
Proc WindowGetPos(&xLow, &yLow{,&xHigh, &yHigh{, scr%{, rel%}}});
```

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 or that returns a docking code for a docked control bar if yHigh is returned as 0.

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 A real variable that is set to the position of the bottom edge of the window or to 0 if the window is docked.

scr%   Optional screen selector, see Window() for details.

rel%   Optional screen selector mode, see Window() for details.

See also:Window(), WindowDuplicate(), WindowSize(), WindowVisible()

WindowSize()

This resizes the current view without changing the top left-hand corner position. Setting a negative size causes no change. Setting a size less than the minimum or greater than the maximum allowed sets the appropriate limit. There are no errors from this function.

```
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.

You can also use this to resize the application window or a dockable control bar when it is floating; use `App()` to get the handles. For control bars, if `width` is greater than zero, its sets the width, otherwise `height` is used to set the height. If the control bar can be resized, it will use the width or the height and will calculate the other dimension itself.

See also:`App()`, `Window()`, `WindowGetPos()`, `WindowTitle$()`

WindowTitle$()

This function gets and sets (where allowed) the current window title. Most windows can return a title. If you change a title, dependent window titles change, for example, cursor windows belonging to time views track the title of the time 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()

This function is used to get and set the visible state of the current window. 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 under Windows, hidden on the Macintosh. An iconised window can be sent data, sized and so on; the result is not visible. This will dock a dockable window at its last docked position.

     3   Maximise, make it as large as possible or float a dockable window.

     4   Application window only; extend over all available desktop monitors.

Returns   The window state prior to this call.

See also:`FrontView()`, `Window()`, `WindowDuplicate()`, `WindowGetPos()`

XAxis()

This turns on and off the x axis of the current view and returns the state of the x axis.

```
Func XAxis({on%});
```

on%   Set the axis state. If omitted, no change is made. 0=Hide axis, 1=Show axis.

Returns   The axis state at the time of the call (0 or 1, 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.

See also:`XAxisMode()`, `XAxisStyle()`, `XHigh()`, `XLow()`, `XRange()`

XAxisAttrib()

This function controls the choice of logarithmic or linear axis in result and XY views and control over the automatic adjustment of axis units in time, result and XY views. This command is equivalent to the controls at the bottom of the X Axis dialog.

```
Func XAxisAttrib({flags%});
```

flags% A value of 0 sets a linear axis with no auto-adjust of units for high zoom. Add 1 for a logarithmic axis. Add 2 to display powers on the logarithmic axis. Add 4 to cause a linear axis to auto-adjust its units at high zoom around 0. Add 8 to scale units using SI prefixes. 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()

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:
  - 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 x axis style and the major and minor tick spacing for views with an x axis. Spacing values that would cause illegible or unintelligible axes are stored but not used unless the axis range or scaling changes to make the values useful.

```
Func XAxisStyle({style%{, nTick%{, major}}});
```

style% In a time view, set 1 for an axis in seconds, 2 for hours minutes and seconds. 3 for time of day, 4 for milliseconds. Result and XY views allow style 1; 4 is also allowed if the x axis units are "s" or "seconds". Omit style% or use 0 to leave the style unchanged. -1 returns the number of minor divisions or 0 for automatic. -2 returns the major tick spacing or 0 for automatic spacing.

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 sets the major tick spacing. Values less than or equal to 0 select automatic major tick spacing.

Returns If style% is positive or omitted it returns the style at the time of the call. See the description of style% for negative values.

See also:XAxis(), XAxisMode(), XHigh(), XLow(), XRange(), YAxisStyle()

XHigh()

This returns the x axis value at the right hand side of the current time, result or XY view, the end of the time range to process for a spike shape view or the line number past the end of the screen for a text view. Use with other views causes a fatal script error.

```
Func XHigh()
```

Returns In a time view or spike shape dialog, the result is in seconds. In a result view, it is in bins and can be fractional. In a text view, it is in lines.

See also:Draw(), XRange(), BinToX(), XToBin(), XLow()

XLow()

This returns the x axis value at the left hand side of the current time, result or XY view, the first visible line in a text view or the start of the time range to process for a spike shape view. It is a fatal error to use this in an inappropriate view.

```
Func XLow()
```

Returns In a time view or a spike shape dialog, the result is in seconds. In a result view, the result is in bins and can be fractional. In a text view, it is in lines.

See also:Draw(), XRange(), BinToX(), XToBin(), XHigh()

XRange()

This sets the x axis range to display in a time result or XY view in x axis units (not bins for a result view). Unlike `Draw()`, the view does not update immediately; updates wait for the next `Draw()`, `DrawAll()`, `Yield()` or some interactive activity. It also sets the time range to process in a spike shape window.

```
Proc XRange(from {,to});
```

from   The left hand edge of the view in x axis units (seconds for a time view). You can set `from` to -1 in a time view that is sampling or re-running to make the view scroll automatically to show the most recent data at the right-hand edge.

to   The right hand edge of the view. If omitted, the view stays the same width.

Values are limited to the axis range for time and result views; there is no limit in an XY view. Without `to`, it preserves the width, adjusting `from` 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 scroll bar and controls.

```
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 gets and sets the x axis title in a result or XY view. In a time view, it has no effect and returns an empty string. The window updates with a new title at the next opportunity.

```
Func XTitle$({new$});
```

New$ If present, this sets the new x axis title in a result view.

Returns The x axis title at the time of the call.

See also:ChanTitle$()

XToBin()

In a result view, this converts between bin numbers and x axis units; in a time view, it converts between time in seconds and the underlying Spike2 time units.

```
Func XToBin(x);
```

x   An x axis value. If it exceeds the x axis range, it is limited to the nearer end.

Returns In a result view it returns the bin position that corresponds to x. In a time view, it converts seconds to the underlying time units used for the file.

See also:BinToX()

XUnits$()

This function gets the units of the x axis. You can also set the units in a result or XY view. The window will update with the new units at the next opportunity.

```
Func XUnits$({new$});
```

New$ If present, this sets the new x axis units in a result view.

Returns The x axis units at the time of the call.

See also:ChanUnits$()

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 was added at version 5.04 and 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:XYDelete(),XYJoin(),XYRange(),XYSetChan(),XYSize(),XYSort()

XYColour()

This gets or sets a channel line colour or a channel fill colour in the current XY view.

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

chan%   A channel number in the current XY view. The first channel is number 1.

col%   The index of the colour in the colour palette. There are 40 colours in the palette, numbered from 0 to 39. If omitted or -1, there is no colour change.

item%   Set 1 for the line colour, 2 for the fill colour. Taken as 1 if omitted.

Returns The colour index in the colour palette at time of call or a negative error code.

See also:Colour(),XYDrawMode(),XYJoin(),XYKey(),XYSetChan()

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. The first channel is number 1.

Returns The number of data points in the channel or a negative error code.

See also:XYAddData(),XYDelete(),XYInCircle(),XYInRect(),XYSize()

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. The first channel is number 1.

first%   The zero-based index of the first point to delete. Omit to delete all points.

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()`, `XYCount()`, `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. The first channel is number 1. This is ignored when which% is 5, as all XY channels share the same axes. -1 can also be used, meaning all channels.

which%   The drawing parameter to get or set in the range 1 to 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 in points (units of approximately 0.353 mm). 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 in points. 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 draw mode or axis expanding mode. If omitted, no change is made.

Returns The value of the relevant channel draw mode or axis expanding mode at the time of the call or a negative error code.

See also:`XYAddData()`, `XYColour()`, `XYCount()`, `XYDelete()`, `XYGetData()`, `XYInCircle()`, `XYInRect()`, `XYJoin()`, `XYKey()`, `XYRange()`

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. The first channel is number 1.

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, 0 is used.

last% The zero-based index of the last data point returned, used when x and y are arrays. If omitted or 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. If the x or y arrays are not big enough to hold all the data points from first% to last%, the return value is the array size. If x or y are not arrays, if a data point with index first% exists, 1 is returned.

The index number of a data point depends on the current sorting method (see XYSort()).

See also:XYAddData(), XYCount(), XYDelete(), XYInCircle(), XYInRect(), XYJoin(), XYRange(), XYKey(), XYSetChan(), XYSize(), XYSort()

XYInChan()

This function returns the number of data points in a channel of the current XY view that lie inside another channel that is treated as a joined up shape. The points in the channel that defines the shape are not sorted (regardless of the sorting mode set for that channel); they are always considered in the order in which the points were added to the channel.

For a data point to lie inside a shape, we count the number of times that a line drawn from the data point to infinity in any direction crosses a line of the shape. Points that are inside have an odd crossing count; points outside have an even count. This is obvious for simple shapes, but less so for complex ones. In the example, the shaded sections are the inside and the non-shaded are the outside. Points that lie exactly on the boundary may be inside or outside; however, if you have a set of shapes that exactly tessellate to fill an area, any point in that area will be in only one of the shapes
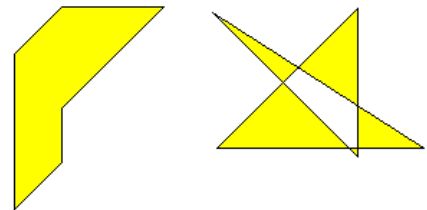
```
Func XYInChan(chan%, shCh%{, list%[]});
```

chan% A channel number in the current XY view that defines the points.

shCh% The channel in the current XY view that defines the shape that the points must be inside.

list% An optional integer array that is returned holding the indices of the data points that were inside. If the array is too small, the function returns the number that would have been returned had the array been large enough. The index values returned are the index positions in the current sort mode for the channel. You can read the points back with XYGetData().

Returns The number of data points inside the shCh% channel or a negative error code.

See also:XYAddData(), XYInCircle(), XYInRect(), XYJoin(), XYRange()

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{, list%[]});
```

chan%   A channel number in the current XY view. The first channel is number 1.

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.

list%   An optional array returned holding the indices of data points in the current sort mode that were inside. If the array is too small, the function returns the number that would have been returned had the array been large enough. You can read the points back with XYGetData().

Returns The number of data points inside the circle or a negative error code.

See also:XYAddData(),XYInChan(),XYInRect(),XYJoin(),XYRange()

XYInRect()

This function returns the number of data points in a channel of the current XY view that lie inside a rectangle. To be inside, a data point must lie between the low rectangle coordinate up to, but not including the high coordinate. This is so that two rectangles with a common edge will not both count a data point on the boundary.

```
Func XYInRect(chan%, xl, yl, xh, yh{, list%[]});
```

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.

list%   An optional array returned holding the indices of data points in the current sort mode that were inside. If the array is too small, the function returns the number that would have been returned had the array been large enough. You can read the points back with XYGetData().

Returns The number of data points inside the rectangle or a negative error code.

See also:XYInChan(),XYInCircle(),XYGetData(),XYJoin(),XYRange()

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), filled or filled and framed.

```
Func XYJoin(chan% {,join%});
```

chan%   A channel number in the current XY view. The first channel is number 1. -1 is also allowed, meaning all channels.

join%   If present, the new join method of the channel. Omit for no change.

    0   Not joined by lines (this is the default joining method)
    1   Joined by lines. The line styles are set by XYDrawMode().
    2   Looped. Joined by lines and the last point is connected to the first.
    3   Not joined by lines, but the channel data is filled with the channel fill colour.
    4   Joined by lines and filled with the channel fill colour.
    5   Histogram. Consecutive points set the bin edges, the first sets the bin height.

Returns The joining method at the time of the call or a negative error code.

See also: XYColour(), XYDrawMode(), XYKey(), XYSetChan(), XYSort()

XYKey()

This gets or sets the display mode and positions of the channel key for the current XY view. The key displays channel titles (set by `ChanTitle$()`) and drawing symbols of all the 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$(), XYColour(), XYDrawMode(), XYJoin(), XYSetChan()

XYOffset()

This command sets and gets a channel offset. The offset moves the origin of the x,y co-ordinate system when the channel is drawn. Offsets are set in the current axis units. These are the channel units for a linear axis and are log(channel units) for a logarithmic axis.

```
Proc XYOffset(chan%, x, y{, opt%});
Proc XYOffset(chan%, &x, &y{, opt%});
```

chan%　A channel number in the current XY view. The first channel is number 1.

x,y　　Used when setting the channel offset. The values are in axis units.

&x,&y　Used when returning the offset. The return values are in axis units.

opt%　　Omit or set to 0 to set the offset. Set to -1 to return the offset.

See also: XYRange()

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%, &xL, &yL, &xH, &yH{, offs%});
```

chan%　A channel number in the current XY view or -1 for all channels or -2 for all visible channels. The first channel is number 1.

xL,yL　Returned holding the smallest x and y co-ordinates in the channel(s).

xH,yH　Returned holding the largest x and y co-ordinates in the channel(s).

offs%　If present and non-zero, channel offsets are taken into account, otherwise they are ignored when calculating the data range.

Returns 0 if there are no data points, or the channel does not exist, 1 if values found.

See also: XYCount(), XYGetData(), XYInCircle(), XYInRect(), XYOffset()

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 modify 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 1 to 256. Spike2 creates the first channel automatically when you open a new XY view with `FileNew()`. If `chan%` is not 0, it must be the channel number of an existing channel to modify or -1 to modify all channels.

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 are deleted to make space for the newer 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; 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 also connect the first and last data points to form a loop.
3   Not joined by lines, but the channel data is filled with the channel fill colour.
4   Joined by lines and filled with the channel fill colour.
5   Histogram. Consecutive points set the bin edges, the first sets the bin height.

col%   If present, this sets the index of the colour in the colour palette to use for this channel. There are 40 colours with indices 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. Use the `ChanColourSet()` command to set arbitrary RBG colours.

Returns   The highest channel number that was affected or a negative error code. When you create a channel, the value returned is the new channel number.

See also:`XYColour()`, `XYDelete()`, , `XYJoin()`, `XYSize()`, `XYSort()`

XYSize()

This gets and sets the limits on the number of data points in the current XY view. Each channels can be set to 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. The first channel is number 1.

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 are deleted to make space for the newer 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(), XYCount(), XYDelete(), 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 in the current XY view. The first channel is number 1.

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(), XYJoin(), XYKey(), XYRange(), XYSetChan(), XYSize()

YAxis()

This gets and sets the visibility of y axes in the current 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:Optimise(), YAxisMode(), YHigh(), YLow(), YRange()

YAxisAttrib()

This function controls the choice of logarithmic or linear axis in result and XY views and control over the automatic adjustment of axis units in time, result and XY views. This command is equivalent to the check boxes at the bottom of the Y Axis dialog.

```
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 a logarithmic axis. Add 2 to display powers on the logarithmic axis. Add 4 to cause a linear axis to auto-adjust its units at high zoom around 0. Add 8 for SI unit prefixes for scaling. Omit this argument for no change to the attributes.

Returns The sum of the current flags set for the first specified channel.

See also: YAxis(), YAxisMode(), YAxisStyle(), YHigh(), YLow(), YRange()

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 nth 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%{, hal%{, hsp%{, hvp%}}}});
```

mode% Optional argument that controls how the axis is displayed. If omitted or negative, no change is made. Positive values are the sum of:

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.
8192	Horizontal text for the title and units.

hal%   Horizontal label alignment: 0 for centred, 1 for edge, -1 or omit for no change.

hsp%   Horizontal label character space in the range 2-17. Set -1 or omit for no change.

hvp%   Horizontal label vertical position: 0 for centred, 1 for top, 2 for bottom. Set -1 or omit for no change.

Returns If mode% is positive, omitted or -1 it returns the mode% value at the time of the call or a negative error code. Set mode% to -2 to return the current value of hal%, -3 for hsp% or -4 for hvp%.

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.

See also:YAxis(), YAxisMode(), YHigh(), YLow(), YRange(), XAxisStyle()

YHigh()

This function returns the upper y axis limit for the y axis of a channel in a time, result or XY view. The return value ignores the grouped or locked state of the channel.

```
Func YHigh(chan%);
```

chan%   The channel number (use 0 for an XY view).

Returns The value at the appropriate end of the axis.

See also:YAxisLock(), YRange(), ChanScale(), ChanOffset(), Optimise()

Yield()

This function suspends script operation for a user defined time and allows Spike2 to idle. During the idle time, invalid screen areas update, you can interact with the program and Spike2 has the opportunity to do housekeeping such as releasing temporary system resources. 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 system marking Spike2 as "not responding".

```
Func Yield({wait{, allow%}});
```

wait   The minimum time to wait, in seconds. If omitted or 0, Spike2 gives the system one idle cycle. If greater than 0 and there is still waiting time left after an idle cycle completes, other processes are given the opportunity to run between idle cycles. 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(), MaxTime(), Seconds(), Toolbar(), YieldSystem()

YieldSystem()

The `YieldSystem()` command causes Spike2 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 Spike2 input queue. It has no effect on sampling, which runs in a separate thread. Unlike `Yield()`, it does not allow Spike2 to idle.

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. Spike2 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.

```
Proc YieldSystem({t});
```

t   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 `t` values greater than 0, the wait is ended by messages; keyboard and mouse activity, timers for screen updates and the like cause messages. If `t` is 0 or omitted, the current time slice is surrendered, but if Spike2 is the highest priority task, it will be re-scheduled immediately. Negative values suspend Spike2 for `-t` seconds regardless of messages.

`YieldSystem()` with `t` values greater than 0 returns immediately if there are messages in the input queue. Unless you allow Spike2 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);
```

gives other processes a chance to run. `t` times up to 0.05 reduce the CPU usage. Larger values have little additional effect as timer messages end the wait early. To give as much system time as possible to other tasks without allowing Spike2 to idle, use:

```
YieldSystem(-0.001);
```

In this case, increasing the time to –10.0 will have an effect; Spike2 will feel completely unresponsive until the time period has elapsed.

See also:`Interact()`, `DlgShow()`, `Seconds()`, `Toolbar()`, `Yield()`

YLow()

This function returns the lower y axis limit for the y axis of a channel in a time, result or XY view. The return value ignores the grouped or locked state of the channel.

```
Func YLow(chan%);
```

chan%   The channel number. Use 0 for an XY view.

Returns The value at the appropriate end of the axis.

See also:`YAxisLock()`, `YRange()`, `ChanScale()`, `ChanOffset()`, `Optimise()`

YRange()

This sets the displayed y axis range for a channel. It has no effect if the current channel display mode has no y axis. If the y range changes, the display updates at the next opportunity. If you omit `low` and `high`, Spike2 displays the "full" range of the channel, equivalent to the Show All button in the Y Axis dialog. In a Spike Shape dialog, the value of `low` or `high` farthest from zero sets the displayed range.

```
Proc YRange(cSpc, low, high});
```

cSpc   A channel specifier for the source channels or -1 for all, -2 for visible and -3 for selected channels.

low   The value for the bottom of the y axis. If omitted, and the channel type has a known range, Spike2 sets `low` and `high` to suitable limits. For example, for a waveform channel displayed as a waveform the limits are those set by the 16-bit nature of the data. For a sonogram channel, the limits are 0 and half the sampling rate. For an event channel, the limits are 0 and the maximum sustained event rate set in the sampling configuration dialog.

high   The value for the top of the y axis. If `low` and `high` are the same, or too close, the range is not changed. It is an error to supply `low` and omit `high`.

See also: YAxisLock(), YHigh(), YLow(), ChanScale(), Optimise()

ZeroFind()

Find a zero (root) of a user-defined continuous function using Brent's method. You must supply a range to search that contains a zero. The method is iterative with each iteration reducing the search range. Iterations stop when a zero is found, the search range becomes smaller than a supplied tolerance or an iteration limit is reached.

```
Func ZeroFind(&x, f(x), a, b{, tol{, maxIt%}});
```

x   A real variable that is returned with a value that is within `tol` of a position for which `f(x)` is zero.

f   The name (no brackets or argument) of a user-defined function that takes a single real argument and that returns a real value.

a,b   These values define a range to be searched for a zero. `f(a)` and `f(b)` must both be non-zero and must have different signs.

tol   This specifies how close the returned `x` must be to the exact result and must be positive. If omitted or zero, the tolerance is set based on the root position and floating point precision.

MaxIt% The maximum number of iterations of the algorithm in the range 1 to 200. If omitted, a maximum of 100 iterations are set.

Returns The number of iterations left after the tolerance is achieved, or 0 if it was not, or -1 if the initial range (`a`, `b`) does not include a zero.

Example   The following example calculates the roots of the equation $x^2-2=0$.

```
func root2(x) return x*x-2 end;

var r1,r2;
ZeroFind(r1, root2, 0, 2);
ZeroFind(r2, root2, 0, -2);
printlog("Roots at %g and %g\n", r1, r2);
```

6 Translating DOS scripts

Introduction

This section is for Spike2 for DOS users who need to move scripts to the Spike2 for Windows environment. If you do not need to translate old scripts you can skip this information. We provide a translator that will do most of the hard work required for the changes, but it cannot cope with all possible scripts and it cannot translate keywords for which no equivalent exists.

To use the translator, open the DOS script as a text file and select the Script menu Convert DOS Script option. This opens a new script window for the output (plus any notes about changes you may need to make) and sends a summary of the notes to the Log window. The translator marks the output with notes like:

```
DrawMode(-1, mode+1);  'Original was "DRAWMODE mode"
'*** Note 7: No exact equivalent of DRAWMODE
'*** The effect of the new DRAWMODE may not be exactly the same. We have
'*** selected all channels and set a mode, please check it.
```

Because the mode is held in a variable the translator has no way of knowing what this will be at run time or what type of view will be active. It has made the best guess it can at the required translation, but is warning that you should inspect the result to be sure that it is right. If the same warning is required later in the script, the message is just:

```
'*** Note 7: No exact equivalent of DRAWMODE
```

Please don't expect the result to be perfect. It is our experience that the more effort went into the DOS script to make it look good on screen, the less successful the result.

What will translate

Spike2 for DOS supports some 146 keywords and 14 operators. Because many of the keywords use numeric codes to make one keyword perform several different functions, there are some 200 different commands to translate. The vast majority translate exactly without any problem. All the maths functions, use of variables, strings and arrays and PROCs translate without any problem, as do expressions using these items.

What will not translate

Spike2 for DOS has built into it a structure of 6 views that can hold time or result views plus text and graphic output streams. It is the mapping of this architecture to a windowed environment that causes most of the problems in translating a script.

We cannot translate concepts that have no equivalent in the windowed environment. For example, moving and drawing lines anywhere on the screen. Scripts that contain syntax errors cannot translate! It is quite common for the translator to find errors in scripts, usually in IF or DOCASE statements in branches that were never executed. If the translator complains about your script, correct the errors and try again.

The keywords that do not translate at all come into the following categories:

- Keywords that make little difference to the functioning of a script or are there because of lack of memory in DOS or are intrinsic to the DOS system or are related to debugging: DUMPM, FREEMEM, NUMVAR, NUMARR, NUMSTR, SHOWVARS, PARAMSTR. The lack of these is usually not a problem.

- Keywords associated with user-defined drawing of lines and/or boxes in views and overdrawing of data. These are the ones that will cause most problems. We will address these in future releases: BOX, DRAWTO, DRAWR, MOVER, OVERDRAW, SETAXES. You may be able to work around this using XY views.

- Keywords associated with running multiple scripts from one script to compensate for the lack of space (20,000 characters) allowed in the DOS scripts. From version 3, the only limit to script size built into the program is that a script must be less than 32,000

lines long. We have not yet decided on the best way to produce large multi-file scripts. Missing keywords are: EXECUTE, EXECUTED, GLOBAL. Note that you can have multiple scripts loaded at a time, so some of the need for these keywords is reduced. From version 4.08, the ScriptRun() command lets you set a script to run after the current script ends. From 6.04 you can use #include.

- Keywords associated with plotting and to some extent printing. Plotting is translated to printing as much as possible. There is no exact equivalent of export in HPGL or PIC format, but there is the far superior (in most cases) option of copy to the clipboard. PLOTTO is not supported at all.

- Finally there is the SETPLACE command. Please let us know if you use it.

Everything else either translates completely, or translates with some worries about compatibility that are marked with notes in the output.

General translation issues

This section describes how the translator copes with problems in the translation, and also explains the extra script variables and functions declared to emulate the original Spike2 for DOS environment.

:= In the DOS version := is treated as an operator, and has a value. In the windowed version, := is not an operator. Instead it indicates an assignment statement. Use of := as an operator is flagged as an error, for example:

```
a:=b:=3.0;         'This is legal in the DOS version
b:=3.0; a := b;    'You must edit the original to this
```

The following is also not translatable (and is often a mistake):

```
if a := b ...      'Allowed in the DOS version
a := b; if a...    'You must edit the original to this
```

Very few scripts make use of the assignment operator in this way.

+-*/... Other operators (+-*/%&| and comparisons) differ because in the new script operators have an order of precedence. This means that 4+5*6 is 34 in the new script, not 54 as in the DOS script. Because of this all expressions are analysed and rewritten to insert and remove brackets as required.

Codes Many DOS script commands use numeric codes to indicate the operation, for example STRFUNC. The new script avoids the use of codes as much as possible to make the commands readable, and because the length of a command name does not slow down the script at run time. For example compare i% := InStr(text$, find$); with STRFUNC 4 text$ find$ I. If you use a variable in place of a code we cannot translate.

Clear This keyword is not translated. If you intend to clear out text you could try using SelectAll();EditClear().

Drawing The commands associated with drawing (BOX, DRAWTO, DRAWR, MOVER and to some extent MOVETO) have no equivalent.

ESCAPE This keyword has no real equivalent and is translated as 0. You can get exactly the same effect by adding an extra Cancel button to an Interact bar, testing the result from the DlgShow() command or by setting an escape button in the Toolbar() command. However, such transformations are beyond the capabilities of the script converter, so you must do them yourself.

Execute This is not translated. Executed is translated as 0.

File   The new file command does not automatically support the use of numbers as part of a name. We add `Func File()` to emulate this and sort out the views.

FRAME   This is ignored in `ON/OFF` and treated as 1 in expressions.

Global   This is translated as `var`.

Help   This is translated, but most unlikely to be of any use unless you write your own help files.

names   The new script language defines many more commands than the DOS version. These commands have descriptive names that may well clash with names your script used. If a name clash is detected we add `QQ` to the end of the original name. The translator also preserves your capitalisation of all user-defined names (though the system is not case-sensitive).

NewEvent The memory channel functions have more functionality than the `NEWEVENT` buffer. The variable `neqq%` holds the memory channel number used to emulate both `NEWEVENT` and `EVENT`. The memory channels behave just like any other channel, so you must be in the correct time view to use them.

Normal   If your script uses this command we add `Proc Normal(v%)` to emulate it.

Numxxx   The `NUMARR`, `NUMVAR` and `NUMSTR` keywords are ignored.

Overdraw This keyword is not supported. It is translated as 0 in expressions. You can overdraw channels in time and result view, but this does not have the same effect. You may also be able to emulate overdrawing with an XY view. The overdraw WaveMark mode will overdraw Spike shapes for you. Version 6 added 3D overdrawing that may be closer to emulating this keyword.

PrintTo   This keyword does not exist, so we add our own to emulate it. We also add the variable `pqq%` to hold the view handle of the view to which printing to a file should be sent. Any printing that previously went to the screen is now routed to the Log window.

Print   When printing text, you must now specify new lines in the output with the `\n` escape sequence. The translator adds this to the end of format strings unless it detects that output is to a device that is specified by an expression that starts with a minus sign. This will work in *almost* all cases, but you should check. The `%d` format now applies to integer output, use `%f` for reals. The translator attempts to do this for you by looking for possible numeric format expressions in all literal strings. Please check all strings in your script that contain `%` to make sure that the translator hasn't corrupted them.

Plot   The difference between printing and plotting is blurred in the windowed environment. Plotting calls are translated to printing calls and the plotter control calls are not supported. To export an image you can copy to the clipboard or save the view as a picture.

Process   There is a `Process()` command in the new script, but unlike the DOS script where the current view must be a time view to use `Process`, in the new script you must be in the result view (this is because there can be multiple result views simultaneously attached to a time view). To get round this we use `ProcessQQ()` to emulate the old situation.

ScreenL The `ScreenL` and `ScreenR` keywords translate to `XLow()` and `XHigh()`. However, users can scroll result views to fractional bin positions. In the windowed version, do not assume that the value returned by `XLow()` or `XHigh()` in a result view is integral.

Setxxx If you create result views, we call `Proc CreateView(v%, vh%),` where `v%` is the old view number (1 to 6) and `vh%` is the handle for the new view. This takes care of linking result views and time views together (in DOS you could have only one result view linked to a time view).

SetAxes This is not translated. You can draw your own images using the XY views, but the mapping between the DOS drawing commands and the Windows ones is not simple and it is probably better done by hand.

TITLE This is ignored in `ON/OFF` and treated as 1 in expressions.

View To emulate the 6 views we add `Proc ViewQQ()` and a two dimensional array `vqq%[7][2]` to hold the view handles of the 6 views and the handle of any result view attached to a time view for `ProcessQQ().` We also add a variable `view%` to hold the current view number (1-6). DOS scripts all assume that there is a time window, so if your script uses views, a call to `ViewQQ(0)` is made at the start to initialise the system. If the current view is already a time view it is used, otherwise you must name a file.

After translating Once Spike2 has translated your script, and you have checked that it runs we would suggest that you consider reworking it so that it no longer depends on the old architecture of 6 views. This usually means declaring variables at the top of your script to hold handles to the views you will use and removing the `vqq%[][]` array, the `view%` variable and `Proc ViewQQ().`

You should also examine the code for places where the script converter has been verbose. For example, `DRAW` becomes `FrontView(view());Draw();` to make sure that a drawn view is visible. Very often you can delete `FrontView(view());` as this is only needed if the current view is hidden or covered by other windows.

—7— XY views

Introduction   XY views have a wide variety of uses, from displaying user-defined graphs to drawing pictures. XY views have the following features:

- One x axis and one y axis shared between all data channels in the XY view, so all the channels share the same space, so you can overdraw one channel with another.

- Up to 256 data channels allowed in the view.

- Each data channel is a list of (x,y) data points. The number of data points in a channel is limited only by available memory and drawing time. However, you can limit the number of data points on a channel, in which case new data points replace the oldest data points.

- The data points can be drawn with markers at each data point. The range of marker styles currently includes: dots, boxes, plus signs, crosses, circles (Windows NT only), triangles, diamonds, horizontal lines and vertical lines. The size of the markers can also be set, and they can be made invisible.

- The data points can be joined with solid, dotted or dashed lines, and the line thickness can be varied. You can also choose to join the last point in a channel to the first point to make a loop and optionally to fill the resulting shape.

- You can sort the order of the data points in a channel by x, by y or by order of insertion in the channel. This is only important if the data points are joined.

- The colour of the lines and markers can be chosen. If no colour is set, the same colour as for a waveform channel in a time view is set.

There are several example scripts included with Spike2 that illustrate some of the uses of XY views. You will find them in the `scripts` folder:

`FFTWater`	This draw a "waterfall" type display showing the variation in the power spectrum of a waveform channel with time.
`DE`	A script that uses a genetic algorithm to fit a double exponential.
`HRaster`	This shows a raster display and a PSTH of event data with the raster drawn above the PSTH.
`Clock`	An analogue clock for your Toolbar idle routine.
`PingPong`	A very silly example indeed.

Creating an XY view   Although you can create an XY view from the File New menu command, the usual way to generate XY views is with the Analysis menu Measurements command or with the script language. The script language can generate an XY view for general use with the `FileNew()` command and as the target for measurements with a `Process()` command with the `MeasureToXY()` command. See the documentation for `MeasureToXY()` for an example.

The following assumes that you have some familiarity with the script language. An XY view always has at least one data channel, so when you create a view, you also create a channel. The following script code shows you how to make an XY view:

```
var xy%;                    'handle for the XY view
xy% := FileNew(12,1);       'type 12=XY, 1=make visible now
```

If you want to add additional channels you can do this using the `XYSetChan()` command. You can also use this command to set a channel to a particular state. The following sets channel 1 (the first channel) to show data points joined by lines with no limit on the number of data points, drawn in the standard colour:

```
XYSetChan(1, 0, 0, 1);      'chan 1, no size limit, no sort, joined
XYDrawMode(1, 2, 0);        'set a marker size of 0 (invisible)
```

To add data points to a channel you use the XYAddData() command. You can add single points, or pass an array of x and y co-ordinates. The following code adds three points to draw a triangle:

```
XYAddData(1, 0, 0);       'add a point to channel 1 at (0, 0)
XYAddData(1, 1, 0);       'add a point at (1,0)
XYAddData(1, 0.5, 1);     'add a point at (0.5, 1)
```

You will notice that the result of this draws only two sides of a triangle. We could complete the figure by adding an additional data point at (0,0), but it is just as easy to change the line joining mode to "Looped", and the figure is completed for you:

```
XYJoin(1,2);              'set looped mode
```

Overdrawing data

You can use the XY view to overdraw data. For example, suppose we have a waveform channel and another channel of trigger markers and we want to superimpose the first 100 data points after each trigger. The following example does this. To make it a little easier to see, we have displaced each triggered data sweep slightly to the right and up. We have also turned off the x and y axes.
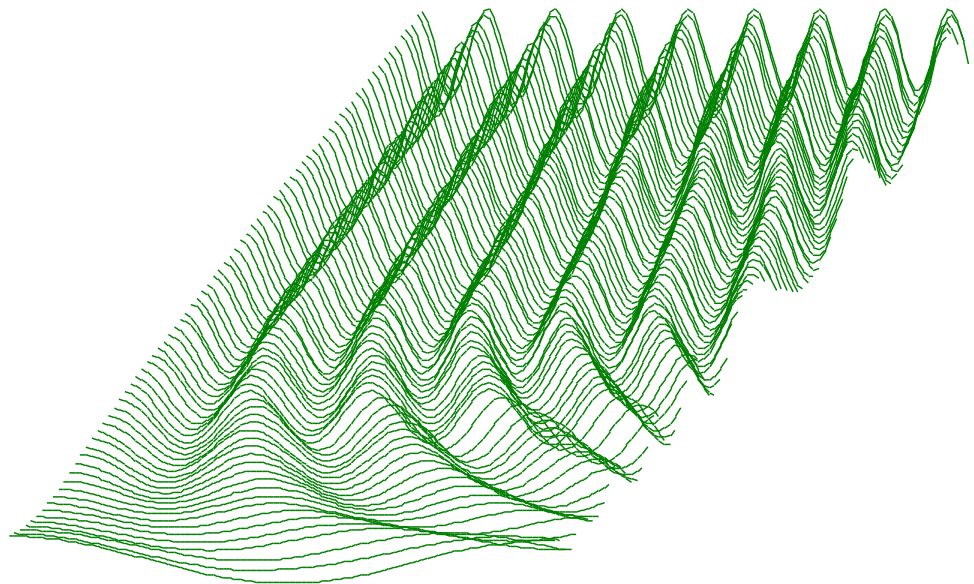
The code required to produce this image is quite short and can be easily adapted to display other types of data. This example expects to run with the demo.smr data file in the Spike2\Data folder created when you installed the program. To run with other files you would need to adjust the channel numbers.

The first line declares variables to remember the time and XY window handles (identifiers) and a variable to hold the number of points read. The second line checks that the current view is a time view and gives up if it is not. The third line remembers the time view handle.

```
var tv%, wh%, np%;        'time view, XY view, points read
if ViewKind() then Message("Not a time view");Halt endif;
tv% := View();            'save view handle
```

The next three lines declare an array to hold 100 data points, a variable to hold the trigger time (and we set it to -1 so we see the first event, even if it is at time 0), and we call the MakeWindow%() function (see below) that creates the XY window for us. We set the first argument to the time interval between data points in out time view so that the x spacing of the data points is correct. The second and third arguments to this function set the distance each "slice" of data is moved to the right and up.

```
var wave[100];                'space for 100 waveform data points
var t := -1;                  'start time for trigger search
wh% := MakeWindow%(BinSize(1), 0.01, 0.5); 'start waterfall view
```

To generate the data we run round a loop (`repeat…until (t<0) or (t>20);`) that finds the time of the next trigger on channel 2 , reads in 100 data points into the array `wave[]` and then as long as the data read correctly, adds the data into the XY window.

```
repeat
   View(tv%);                 'Make time view current
   t := NextTime(2, t);       'Get next trigger time from channel 2
   np% := ChanData(1, wave[], t, MaxTime()); 'get channel 1 data
   if (np% > 0) then AddSlice(wave[:np%]) endif; 'add to picture
until (t < 0) or (t>20);      'until end or we reach 20 seconds
```

The final task is to make the XY window visible and halt.

```
View(wh%).WindowVisible(1); 'make XY window visible
halt;
```

The remaining code can be copied from the `FFTWater.s2s` script. The variables with names starting `WF` are used to remember the initial waterfall settings. There are two functions in this code. `MakeWindow%()` prepares for the waterfall display by creating the XY window and storing the information needed to position the channels. `AddSlice()` takes an array of data points and adds it to the display as an additional channel.

```
'================== Waterfall display code =====================
var WFxInc,WFyInc,WFbinSz,WFSlices%,WFvh%;
'
'xBinSz Width of each bin (sample interval)
'xInc   Add to each slice x co-ords to give waterfall effect
'yInc   Add to each slice y co-ords to give waterfall effect
Func MakeWindow%(xBinSz, xInc, yInc)
WFSlices% := 0;               'no slices yet
WFxInc := xInc;               'save x increment per slice
WFyInc := yInc;               'save y increment per slice
WFbinSz:= xBinSz;             'save data point separation
WFvh% := FileNew(12);         'create a new XY window (hidden)
return WFvh%;                 'return the XY window handle
end;

Func AddSlice(y[])            'Add data to the waterfall
View(WFvh%);                  'select the waterfall view
var ch%:=1;                   'true if this is the first channel
if WFSlices% = 0 then         'if first channel no need to create
   XYSetChan(1,-Len(y[]),0,1);           'set original channel
else
   ch% := XYSetChan(0, -Len(y[]), 0, 1);'create new channel
   if (ch% <= 0) then return ch% endif; 'No more channels
endif;
WFSlices% := ch%;             'number of slices
XYDrawMode(ch%,2,0);          'Hide the markers (set size of 0)
var x[Len(y[])];              'space for x values, same size as y[]
ArrConst(x[], WFbinSz);       'generate x axis values, set the same
x[0]:=(ch%-1)*WFxInc;         'set x offset as first value
ArrIntgl(x[]);                'form the x positions
ArrAdd(y[],(ch%-1)*WFyInc);   'add the y offset to the y array
XYAddData(ch%, x[], y[]);     'add the (x,y) data points
return 1;                     'return >0 means all OK
end;
```

Of course, from Spike2 6.09 you can generate this display in a time view using the `ViewTrigger()`, `ViewOverdraw()` and `ViewOverdraw3D()` commands, or do it interactively with the Display Trigger and View Overdraw dialogs.

New Features in version 7

At version 7.01, there are new extensions to the script language in the toolbar and user-defined dialogs that give the script writer access to the mouse position and left-hand mouse button clicks. To support this, the XYInRect() and XYInCircle() commands can now return a list of point indices that are hit, in addition to the count. There is also a new command, XYInChan() that reports which data points on a channel lie within a polygon defined by another channel. These new commands allow you to use the mouse to select individual points (use ChanPixel() to set the tolerance for a hit), and to draw an arbitrary shape and detect how many data points lie within it.

At version 7.09 XY view channels can now be draw in Histogram mode. In this mode, each consecutive pair of points (in the current sort order) defines one bin of the histogram. The x co-ordinates define the start and end of each bin and the y co-ordinate of the first point defines the bin height. If you have n points, you will see n-1 bins as the final point x co-ordinate defines the width of the last bin. You can also use the new XYOffset() script command to offset the XY view axis origin when drawing a channel.

8   Curve fitting from a script

Introduction

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 $y = f(x, \mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2 …)$ 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.
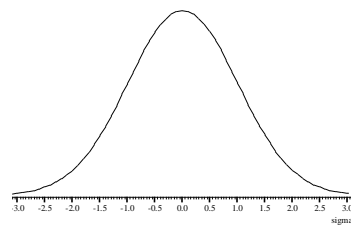
There is a particular error distribution that often occurs, called the Normal distribution. If a set of measurements is normally distributed, with a mean y_m and standard deviation σ, then the probability of measuring any particular value y is proportional to:

$P(y) \propto \exp(-\tfrac{1}{2}(y-y_m)^2/\sigma^2)$

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:

$$\chi^2 = \Sigma_j \ ((y_j - f(x_j, \mathbf{a}_i) \ )^2 / \sigma_j^2)$$

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 if 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 (based on the assumption that the variance is the same for them all and that the model does fit the data), by fitting your model and calculating 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 if the correct model function has been selected, or if 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, $y = \mathbf{a}_0 \exp(-\mathbf{a}_1 x) + \mathbf{a}_2$. We solve these types of problem 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 script 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 by multiplying 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 `nC%` coefficients and the sum of squares of the errors is `errSq`, then use `ArrMul(covar[][], errSq/(nD%-nC%));` 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 coefficient 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 given that the data is correctly modelled by the fitting function plus normally distributed noise?" 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 model does not explain all the variation in the data.

Fitting routines

The `ChanFit...()` family of commands give a script the same capability as the Analysis menu Fit Data command. These commands were added to Spike2 at version 6.

ChanFit() This command associates a fit with a data channel in a time, result or XY view, performs the fit and returns basic fit information.

ChanFitCoef() This gets and sets fit coefficients, coefficient limits and the hold flags. It is equivalent to the coefficient page of the Fit Data dialog.

ChanFitShow() This controls the display of the fitted data.

ChanFitValue() This returns the value at a given x position of the fitted function.

The remaining fitting functions allow you to fit curves to data in arrays:

FitPoly() This fits a polynomial of the form $y = \mathbf{a}_0 + \mathbf{a}_1 x + \mathbf{a}_2 x^2 + \mathbf{a}_3 x^3 \ldots$

FitLine() This fits a straight line to data in a time or result view.

FitLinear() This fits $y = \mathbf{a}_0 f_0(x) + \mathbf{a}_1 f_1(x) + \mathbf{a}_2 f_2(x)\ldots$where the $f_n(x)$ are user-defined functions of *x*.

FitExp() This fits multiple exponentials of the form $y = \mathbf{a}_0 \exp(-x/\mathbf{a}_1) + \mathbf{a}_2 \exp(-x/\mathbf{a}_3)\ldots$

FitGauss() This fits Gaussians of the form $y = \mathbf{a}_0 \exp(-\frac{1}{2}(x-\mathbf{a}_1)^2/\mathbf{a}_2^2) + \mathbf{a}_3 \exp(-\frac{1}{2}(x-\mathbf{a}_4)^2/\mathbf{a}_5^2) + \ldots$

FitSin() This fits multiple sinusoids of the form $y = \mathbf{a}_0 \sin(\mathbf{a}_1 x + \mathbf{a}_2) + \mathbf{a}_3 \sin(\mathbf{a}_4 x + \mathbf{a}_5) + \ldots$

FitNLUser() This fits a user-defined function of the form $y = f(x, \mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2\ldots)$ to a set of data points where the $\mathbf{a}_i$ are constant coefficients to be determined. You must be able to calculate the differential of the function *f* with respect to each of the $\mathbf{a}_i$ coefficients. This is the most general fitting routine, and also the slowest and most complex to use.

GammaP(), GammaQ() Use these functions to calculate the error function *erf(x)*, the cumulative Poisson probability function and the Chi-squared probability function. These are very useful when considering probabilities of fits associated with the normally distributed data.

BetaI() This calculates the incomplete Beta function, which is used to work with the Binomial, Students T and the F distributions.

LnGamma() This calculates the natural logarithm of the Gamma function. It can be used to calculate the log of large factorials and is useful when working with probability distributions.

Index