The Spike2 script language

Version 4

First edition	January 1995
Version 3.00	January 1998
Version 4.00	January 2001
Version 4.03	September 2001
Version 4.04	January 2002
Version 4.05	February 2002
Version 4.06	March 2002
Version 4.07	April 2002
Version 4.08	June 2002
Version 4.09	July 2002
Version 4.10	August 2002
Version 4.14	February 2003
Version 4.16	May 2003
Version 4.18	October 2003
Version 4.19	January 2004

Published by:

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

Telephone:	Cambridge (01223) 420186
	International +44 1223 420186
Fax:	Cambridge (01223) 420488
	International +44 1223 420488
Email:	info@ced.co.uk
Home page:	www.ced.co.uk

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 360 functions pre-programmed, 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

This manual makes no distinction between the two flavours of Spike2 (Windows and Macintosh) except where there is nothing that we can do about it in which case the behaviour is noted as "system dependent". 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. The Macintosh version stops at version 2, so if you use any version 3 or 4 features, your script will not run on a Macintosh.

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.

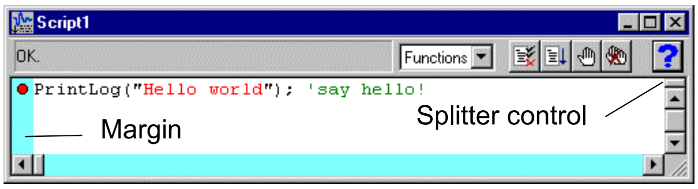
Sources of script information

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 is a separate manual provided with Spike2 that has been used for our Spike2 training courses, held at CED. 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.



```
PrintLog("Hello world"); 'say hello!
```

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

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

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 moving the mouse pointer over the margin on the left of the script and double clicking.

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

Clear all break points

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

Help

This button provides help on the script language. It takes you to an alphabetic list of all the built-in script functions. If you scroll to the bottom of this list you can also find links to the script language syntax and to the script language command 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 version 4 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 Editor settings section of the Edit menu Preferences. 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. If you like to indent your text, you can indent and outdent selected blocks by selecting one or more characters, then use the `Tab` key to indent and the `Shift-Tab` combination to outdent. The text moves by one tab stop. Tab stop sizes are set in the Edit menu Preferences.

Debug overview

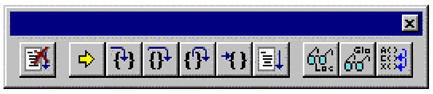
Despite all our attempts to make writing a script easy, and all your attempts to get things right, sooner or later (usually sooner), a script will refuse to do what you expect. Rather than admit to human error, programmers attribute such failures to “bugs”, hence the act of removing such effects is “debugging”.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

To make bug extermination a relatively simple task, 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
- View local and global variables
- Edit variable values
- See how you reached a particular function or procedure
- Set and clear break points

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

Preparing to debug



Unlike most languages, the Spike2 script language does not need any special preparation for debugging except that you must set a break point or include the `Debug();` command at the point in your script at which you want to enter the debugger. When your script reaches the break point or the `Debug();` command, the script stops and the debug toolbar will appear (if it was not already visible). The picture above shows the toolbar as a floating window, but you can dock it to any side of the Spike2 window by dragging over the window edge.

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

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

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

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

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

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

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

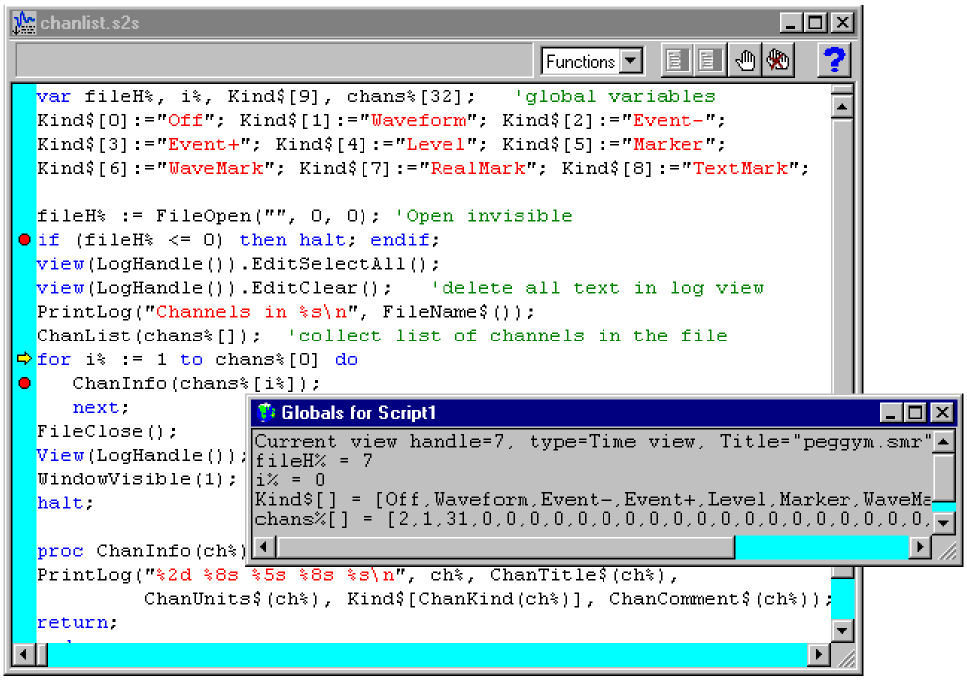
Run the script. This disables the buttons on the debug toolbar and the script runs until in reaches a break point or the end of the script.

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

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

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

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



```
var fileH%, i%, Kind$[9], chans%[32];   'global variables
Kind$[0]:="Off"; Kind$[1]:="Waveform"; Kind$[2]:="Event-";
Kind$[3]:="Event+"; Kind$[4]:="Level"; Kind$[5]:="Marker";
Kind$[6]:="WaveMark"; Kind$[7]:="RealMark"; Kind$[8]:="TextMark";

fileH% := FileOpen("", 0, 0); 'Open invisible
if (fileH% <= 0) then halt; endif;
view(LogHandle()).EditSelectAll();
view(LogHandle()).EditClear();   'delete all text in log view
PrintLog("Channels in %s\n", FileName$());
ChanList(chans%[]);  'collect list of channels in the file
for i% := 1 to chans%[0] do
   ChanInfo(chans%[i%]);
   next;
FileClose();
View(LogHandle());
WindowVisible(1);
halt;

proc ChanInfo(ch%)
PrintLog("%2d %8s %5s %8s %s\n", ch%, ChanTitle$(ch%),
         ChanUnits$(ch%), Kind$[ChanKind(ch%)], ChanComment$(ch%));
return;
```

Globals for Script1

```
Current view handle=7, type=Time view, Title="peggym.smr"
fileH% = 7
i% = 0
Kind$[] = [Off,Waveform,Event-,Event+,Level,Marker,WaveMa
chans%[] = [2,1,31,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
```

The example above shows a script that prompts the user for a data file, opens it and prints a summary of the data channels to the log window. The user set two break points, ran the script then clicked on the Step button five times and clicked the Globals button.

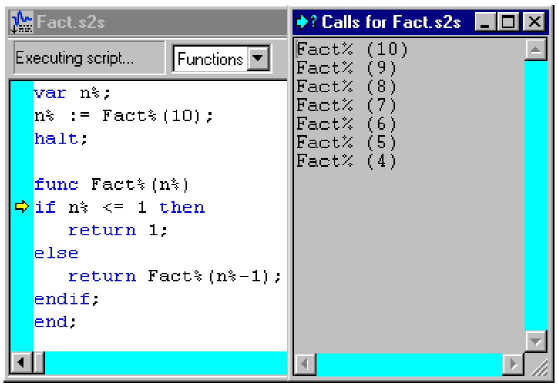
Inspecting variables

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

Call stack

The call stack can sometimes be useful to figure out how your script arrived at a position in your code. This is particularly true if your script makes recursive use of functions. A function is recursive when it calls itself, either directly, or indirectly through other functions. This example implements 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.



```
var n%;
n% := Fact%(10);
halt;

func Fact%(n%)
if n% <= 1 then
   return 1;
else
   return Fact%(n%-1);
endif;
end;
```

```
Fact% (10)
Fact% (9)
Fact% (8)
Fact% (7)
Fact% (6)
Fact% (5)
Fact% (4)
```

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 Spike2 script editor and this size varies with the version of the program. All versions support scripts of at least 32,000 characters. From version 3 there is no editor limit except available memory.

Keywords and names

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

User-defined names can extend up to a line in length. Most users will restrict themselves to a maximum of 20 or so characters. The following keywords are reserved and cannot be used for variables or function names:

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

The `diag`, `trans` and `resize` keywords are not used in version 4; they are reserved for future versions. Further, names of Spike2 built-in functions cannot be redefined as user functions or global variables. They can be used 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 items, for example the weight of an object.

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

Real data type

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

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

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

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

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

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

Integer data type

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

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

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

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

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

String data type

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

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

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

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

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

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

Conversion between data types

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

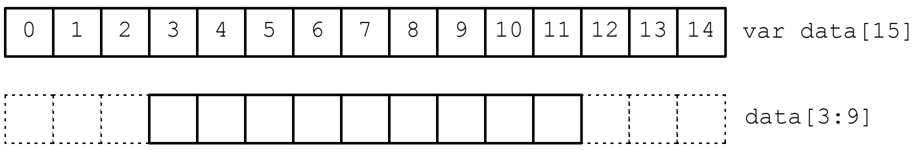
Arrays of data

The three basic types (integers, reals and strings) can be made into one and two dimensional arrays. You declare arrays with the `var` statement and as function arguments. To reference array elements, enclose the array element number in square brackets, for example: `fred[24]`. The first array element is number 0. To reference two dimensional arrays use two sets of brackets: `jim[3][23]`. The current result view is referenced as an array with no name, for example `[0]`, `[1]`, `[2]` for the first three bins.

You specify sub-arrays of an array as `array[start:size]`. The colon inside square brackets, or empty square brackets means that you are defining a sub-array. The number before the colon specifies the start index, the number after the colon specifies the number of elements. If you omit `start`, 0 is used. If you omit `size`, the array up to the end from the start index is used.

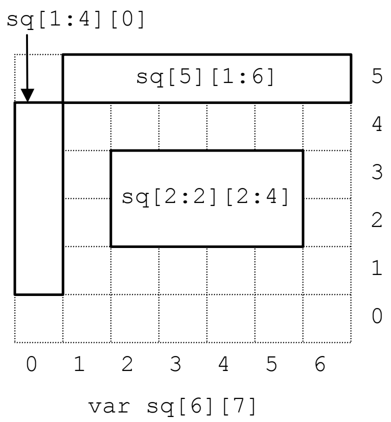
For example, consider the array of real numbers declared as `var data[15]`. This array has 15 elements numbered 0 to 14. To pass the array to a function, you could specify:

`data[3]`	This is a real variable, being the fourth element of the array.
`data[]`	This is the entire array. This is the same as `data[:]` or `data[0:15]`.
`data[3:9]`	This is a sub-array of length 9, being elements 3 to 11.
`data[:8]`	This is a sub-array, being elements 0 to 7.



With a two dimensional array, life is a little more complicated. You can pass a single array element, a one dimensional array, or a two dimensional array. Consider the array `sq[6][7]`. The following possibilities exist:

`sq[a][b]`	a single real number
`sq[a][b:c]`	sub-array of length c
`sq[a:b][c]`	sub-array of length b
`sq[a:b][c:d]`	sub-array dimensions `b` by `d`



This diagram shows how the sub-arrays are constructed. `sq[1:4][0]` is a one dimensional array of 4 elements. This could be passed to a function that expects a one dimensional array as an argument. `sq[5][1:6]` is also a one dimensional array, this time of 6 elements. `sq[2:2][2:4]` is a two dimensional array, with dimensions `[2][4]`.

Result views as arrays

The script language treats a result view as anonymous arrays. To access an individual array 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 sub-array 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
- 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 are not available before version 3.02. `+=` can also be used with strings (`a$+=b$` is the same as `a$:=a$+b$`, but is more efficient).
- A flow of control statement, described below
- A procedure call or a function with the result ignored, for example `View(vh%);`

Comments in a script

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

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

Variable declarations

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

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

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

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

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

Constant declarations

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

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

Expressions and operators

Anywhere in the script where a numeric value can be used, so can a numeric expression. Anywhere a string constant could be used, a string expression is also acceptable. 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	`[], ()`	Array 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
Lowest	`or, xor, bor, bxor`	Logical and exclusive or and bitwise versions

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

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

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

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

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

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

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

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

String operators

	Operators	Names
Highest	+	Concatenate
	`<, <=, >, >=`	Less, less or equal, greater, greater or equal
Lowest	=, <>	Equal and not equal

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

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

Do not confuse assignment := with the equality comparison operator, =. They are entirely different. The result of an assignment does not have a value, so you cannot write statements like `a:=b:=c;` (which was allowed in the MS-DOS version of Spike2).

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$ := "Jane";
alf$ := "alf";
sam := jane$ > alf$;     'Result is 0.0 (a is greater than J)
sam := UCase$(jane$)>UCase$(alf$); 'Result is 1.0 (A < J)
sam := "same" > "sam";   'Result is 1.0
pip% := 23 mod 7;        'Result is 2
jim := 23 mod 6.5;       'Result is 3.5
jim := -32 mod 6;        'Result is -2.0
sue% := jim and not sam;'Result is 0 (jim = -2.0 and sam = 1.0)
pip% := 1 and 0 or 2>1; 'Result is 1
sue% := 9 band 8;        'Result is 8 (9=1001 in binary, 8=1000)
sue% := 9 bxor 8;        'Result is 1
sue% := 9 bor 8;         'Result is 9
```

Mathematical constants

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

Flow of control statements

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

if...endif

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

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

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

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

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

docase...endcase

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`v`	This is the loop variable and may be a real number, or an integer. It must be a simple variable, not an array element.
`exp1`	This expression sets the initial variable value before the looping begins.
`exp2`	This expression is evaluated once, before the loop starts, and is used to test for the end of the loop. If `step` is positive or omitted (when it is 1), the loop stops when the variable is greater than `exp2`. If `step` is negative, the loop stops when the variable is less than `exp2`.

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

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

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

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

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

Functions and procedures

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

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

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

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

Argument lists

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

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

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

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

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

Func SumArr2(data[][])      'Efficient sum of 2d array elements
var n1%,n2%,i%,sum;         'sizes, index and sum, all set to 0
n1% := Len(data[0][]);      'get sizes of arrays...
n2% := Len(data[][0]);      '...so we can see which is bigger
if n1%>n2% then             'choose most efficient method
 for i%:=0 to n2%-1 do sum := sum + ArrSum(data[i%][]) next;
else
 for i%:=0 to n1%-1 do sum := 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 array is the number of channels. The remaining elements are channel numbers. It is an error for the array 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.

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
`BinToX`	Convert from bins or underlying units to x axis units
`Colour`	Get or set the palette entry associated with a screen item
`Dup`	Get the view handle of duplicates of the current view
`Draw`	Draw invalid regions of the view (and set x axis range)
`DrawAll`	Update all invalid regions in all views
`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$`	Gets the file name associated with a window
`FileNew`	Opens an output file or a new text or data window
`FileOpen`	Opens an existing file (in a window)
`FilePrint`	Prints a range of data from the current view
`FilePrintVisible`	Prints the current view
`FilePrintScreen`	Prints all text-based, time and result views
`FileQuit`	Closes the application
`FileSave`	Save a view
`FileSaveAs`	Save a view in variety of formats
`FontGet`	Read back information about the font
`FontSet`	Set the font for the current window
`FrontView`	Get or set the front window on screen
`Grid`	Get or set the visibility of the axis grid
`LogHandle`	Gets the view handle of the log window
`PaletteGet`	Get the RGB colour of a palette entry
`PaletteSet`	Set the RGB colour of a palette entry
`SampleHandle`	Gets the view handle of sampling windows and controls
`View`	Change or override current view and get view handle
`ViewFind`	Get a view handle from a view title
`ViewKind`	Get the type of the current view
`ViewList`	Form a list of handles to views that meet a specification
`ViewStandard`	Returns a window to a standard state
`ViewUseColour`	Get and set monochrome/colour use for view
`Window`	Sets the window size and position
`WindowDuplicate`	Duplicate a time view
`WindowGetPos`	Get window position
`WindowSize`	Changes the window size
`WindowTitle$`	Gets or changes the window title
`WindowVisible`	Sets or gets the visibility of the window (hide/show)
`XAxis`	Get or set 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
`XToBin`	Convert from x axis units to bins or underlying units
`YAxis`	Get or set the visibility of the y axis
`YAxisMode`	Control how the y axis is drawn
`YAxisStyle`	Control y axis tick spacing

Time views

These commands manipulate time views.

`BinToX`	Convert time units to x axis units
`ChanNumbers`	Hide and show channel numbers
`DrawMode`	Get or set display mode for a channel
`EventToWaveform`	Create a waveform channel from events
`ExportChanFormat`	Set channel format for export
`ExportChanList`	Set list of channels for export
`ExportTextFormat`	Set format for text output of channels
`Maxtime`	Maximum x axis value for any channel in the view
`MemChan`	Create a channel in memory
`MemSave`	Write the contents of a memory channel to the data file
`Optimise`	Set reasonable y range for channels with axes
`ReRun`	Get or set the rerunning state of a channel
`ViewTrigger`	Control on-line triggered displays and off-line display stepping
`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$`	Get x axis title
`XToBin`	Convert x axis units to time units
`XUnits$`	Get units of the x axis
`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 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`	Set additional data for sorted rasters (deprecated)
`RasterGet`	Read back raster information
`RasterSet`	Set raster information
`RasterSort`	Get and set raster sort values, replaces `RasterAux()`
`RasterSymbol`	Get and set raster symbol times, replaces `RasterAux()`
`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$`	Get and set x axis title
`XToBin`	Convert x axis value to bin number
`XUnits$`	Get and set units of the x axis
`YHigh`	Get upper limit of y axis
`YLow`	Get lower limit of y axis
`YRange`	Set y axis range

XY views

These commands manipulate XY views. XY views contain up to 256 channels of data and always contain at least one channel. Each channel holds a list of (x, y) co-ordinate pairs that can be displayed in a variety of styles. Most functions that work for views in general will work for an XY view. 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
`XYColour`	Set the colour of a channel
`XYCount`	Return the number of XY data points in a channel
`XYDelete`	Delete one or more data points from a channel
`XYDrawMode`	Control how a channel is drawn
`XYGetData`	Read data back from an XY channel
`XYInCircle`	Count the number of XY points within a circle
`XYInRect`	Count the number of XY points inside a rectangle
`XYJoin`	Get or set the point joining method
`XYKey`	Control the display of the XY view channel key
`XYRange`	Get rectangle containing one or more channels
`XYSetChan`	Create or modify an XY channel
`XYSize`	Get or set maximum size of an XY channel
`XYSort`	Change the sort (and draw) order of a channel

Editing operations

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

`EditClear`	Delete text from a text window at the caret
`EditCopy`	Copy the current selection to the clipboard
`EditCut`	Delete the current selection to the clipboard
`EditPaste`	Paste the clipboard into the current text field
`EditSelectAll`	Select the entire text or cursor window contents
`MoveBy`	Move relative to current position
`MoveTo`	Move to a particular place
`Selection$`	This function returns the text that is currently selected

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

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
`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
`ProgKill`	Kill external process
`ProgRun`	Execute file or shell command
`ProgStatus`	Check the status of an external process

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)
`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

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

Serial line control

These functions let the script writer read to and write from serial line ports on their computer. This feature would normally be used to control equipment during data capture.

`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

Channel commands

These commands manipulate data channels in a time view. They are also used in result views where the channel number is 0 if the result view is one-dimensional.

`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`
`ChanCalibrate`	Calibrate waveform or WaveMark channels from known values
`ChanColour`	Override drawing mode based channel colours
`ChanComment$`	Get or set the channel comment
`ChanData`	Fill an array with a waveform or event times
`ChanDelete`	Delete a channel from a time view
`ChanDuplicate`	Duplicate channels in a time view
`ChanHide`	Make a channel invisible
`ChanKind`	Get the type of a channel
`ChanList`	Get a list of channels meeting a specification
`ChanMeasure`	Make the same measurements as the Cursor Regions dialog
`ChanNew`	Create a new channel for `ChanWriteWave()` to write to
`ChanOffset`	Set waveform and WaveMark value for input of zero
`ChanOrder`	Modify the channel order and y axis grouping
`ChanProcessAdd`	Add new processing option to a waveform or RealWave channel
`ChanProcessArg`	Read or modify an argument of a channel process
`ChanProcessClear`	Delete a process, all processes for a channel, or all processes
`ChanProcessInfo`	Get information about the processes attached to a channel
`ChanPort`	Get the physical sampling port of a data channel
`ChanScale`	Set waveform and WaveMark channel scale factor
`ChanSearch`	Search a channel for a feature (peak, level, slope…)
`ChanSelect`	Select and report on selected state of channels
`ChanShow`	Make a channel visible
`ChanTitle$`	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
`EventToWaveform`	Convert an event channel into a waveform channel
`FitLine`	Fit a straight line to waveform or result channel
`LastTime`	Find the previous item in a channel (and return values)
`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
`Maxtime`	Time of last item on the channel
`MemChan`	Create a channel in memory
`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)

User interaction commands

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

`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
`Yield`	Give idle time to the system; delay for a time

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

`DlgCreate`	Start a dialog definition
`DlgChan`	Define a dialog entry as prompt and channel selection
`DlgCheck`	Define a dialog item as a check box
`DlgInteger`	Define a dialog entry as prompt and integer number input
`DlgLabel`	Define a dialog entry as prompt only
`DlgList`	Define a dialog entry as prompt and selection from a list
`DlgReal`	Define a dialog entry as prompt and real number input
`DlgShow`	Display the dialog, get values of fields
`DlgString`	Define a dialog entry as prompt and string input
`DlgText`	Define a fixed text string for the dialog

The toolbar commands give the user control over the various toolbars and let the script writer link script functions to the script 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
`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

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
`Right$`	Returns the rightmost characters of a string
`Str$`	Converts a number to a string
`UCase$`	Returns upper case version of a string
`Val`	Converts a string to number

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 raster sort values, replaces `RasterAux()`
`RasterSymbol`	Get and set raster symbol times, replaces `RasterAux()`
`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

Array arithmetic

These functions are used with arrays and result views to speed up data manipulation. These functions operate on one dimensional arrays only. Integer arrays can be used where indicated, but beware of overflow. The array arithmetic attempts to fix problems by setting the result element to a (possibly) useful value. The functions all return a negative error code if there is a problem or zero if the function completed without error. You can apply built-in mathematical functions directly on an array. For example, to form the square root of all the elements of array `fred[]` use `Sqrt(fred[])`.

`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
`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

Mathematical functions

The following mathematical functions are built into Spike2. You can apply the arithmetic 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
`Cos`	Cosine of a number or array
`Exp`	Exponential function of a number or array
`Frac`	Remove integral part of a number or array
`Ln`	Natural logarithm of a number or array
`Log`	Logarithm to base 10 of a number or array
`Max`	Finds maximum of array or variables
`Min`	Finds minimum of array or variables
`Pow`	Raise a number or an array to a power
`Rand`	Returns a pseudo-random number
`Round`	Round a real number to the nearest integral value
`Sin`	Sine of a number or array
`Sqrt`	Square root of a number or an array
`Tan`	Tangent of a number or array
`Trunc`	Remove fractional part of number or array

Digital filtering

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

`ArrFilt`	Array arithmetic routine to apply FIR coefficients to an array
`FiltApply`	Apply a set of coefficients or a filter bank filter to a waveform
`FiltAtten`	Set the desired attenuation of a filter in the filter bank
`FiltCalc`	Force coefficient calculation of a filter in the filter bank
`FiltComment$`	Get or set comment for a filter in the filter bank
`FiltCreate`	Create a new filter definition in the filter bank
`FiltInfo`	Retrieve information about a filter in the filter bank
`FiltName$`	Get or set the name of a filter in the filter bank
`FiltRange`	Get the useful sampling rate range for a filter bank filter
`FIRMake`	Generate FIR filter coefficients in an array
`FIRQuick`	Generate FIR filter coefficients with desired attenuation
`FIRResponse`	Calculate frequency response of array of coefficients

Arbitrary waveform output

These commands control the output of waveforms from 1401 memory during data sampling. Waveforms are identified by a key code and up to 10 can be defined.

`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$`	Get waveform information including list of all waveform keys
`PlayWaveLabel$`	Get or set the label associated with each waveform
`PlayWaveLink$`	Get or set the links between waveforms
`PlayWaveRate`	Get or set the desired base wave replay rate
`PlayWaveSpeed`	Scale factor for the wave replay rate - can be used on-line
`PlayWaveStatus$`	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

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$`	Set file name template for automatic file saving
`SampleCalibrate`	Set calibration of waveform or WaveMark channel
`SampleComment$`	Set or get the channel comment
`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
`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
`SampleSequencer`	Set the name of the sequencer file
`SampleSequencer$`	Get the name of the sequencer file
`SampleTextMark`	Adds a channel for text notes
`SampleTimePerAdc`	Set the base ADC conversion interval
`SampleTitle$`	Set and get the title of a channel
`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. There is only one new data file at a time, and these commands refer to it, regardless of the current view.

`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
`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

Discriminator control

These functions control the CED 1401-18 discriminator card.

`DiscrimChanGet`	Get the settings of a discriminator channel
`DiscrimChanSet`	Change discriminator channel settings
`DiscrimClear`	Set all channels to a known state
`DiscrimLevel`	Get or set discriminator thresholds
`DiscrimMode`	Get or set discriminator modes
`DiscrimMonitor`	Get or set the waveform monitor channel
`DiscrimTimeOut`	Get or set the time-out for modes 7 and 8

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$`	Get system name as a string
`Time$`	Get system time in a string in a variety of formats
`TimeDate`	Get system date and time as numbers

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

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%` -3 The program serial number.
-2 The highest `type%` that returns a value.
-1 The program revision multiplied by 100 (version 3.00 and later only).

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

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

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()`, `Window()`

ArrAdd()

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

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

`dest`	The destination array or result view.
`source`	A source array, to add element by element to the destination array. If the arrays are of different sizes, the size of the smaller sets the number of elements copied.
`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];
ArrAdd(fred[],1.0);          'Add 1.0 to all elements of fred
ArrAdd(jim%[2:8],10);        'Add 10 to elements 2-9 of jim%
ArrAdd(fred[],jim%[]);       'Add elements 0-99 of jim% to fred
ArrAdd([],fred[10]);         'Add fred[10] to result channel 1
```

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.

```
Func ArrConst(dest[]|dest%[], source[]|value);
```

`dest`	The destination real or integer array or a result view.
`source`	A source array to copy to the destination array. If the two arrays are of different sizes, the size of the smaller array determines the number of elements copied.
`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.

See also: `ArrAdd(), ArrDiff(), ArrDiv(), ArrDivR(), ArrDot(), ArrFFT(), 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 array that is replaced by an array of differences. The first element of the array is left unchanged.

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

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

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

ArrDiv()

This function divides a real or integer array by an array or a constant. Use `ArrDivR()` to form the reciprocal of an array.

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

`dest`	An array of reals or integers.
`source`	An array of reals or integers used as the denominator of the division.
`value`	A value used as the denominator of the division.
Returns	The function returns 0 if there were no problems, or a negative error code.

The following examples show the effect of the various combinations of arguments:

```
var dest[100], source[100], value, i%;
ArrDiv(dest[],source[]);          'Is equivalent to...
for i%:=0 to 99 do
    dest[i%] := dest[i%] / source[i%];
    next;
```

```
ArrDiv(dest[],value);            'Is equivalent to...
for i%:=0 to 99 do
    dest[i%] := dest[i%]/value;
    next;
```

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

ArrDivR()

This function (Array Divide Reversed) 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`	An array of reals or integers used as the numerator of the division.
`value`	A value used as the numerator of the division.
Returns	The function returns 0 if there were no problems, or a negative error code if there was a problem (for example division by zero or integer overflow).

The following examples show the effect of the various combinations of arguments:

```
var dest[100], source[100], value, i%;
 ArrDivR(dest[],source[]);         'Is equivalent to...
for i%:=0 to 99 do
    dest[i%] := source[i%] / dest[i%];
    next;

ArrDivR(dest[],value);            'Is equivalent to...
for i%:=0 to 99 do
    dest[i%] := value / dest[i%];
    next;
```

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

ArrDot()

This function multiplies one array by another and returns the sum of the products (sometimes called the dot product of two arrays). The arrays are not changed.

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

`arr1`	An array of reals or integers.
`arr2`	An array of reals or integers.
Returns	The function returns the sum of the products of the corresponding elements of the two arrays.

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

ArrFFT()

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:

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

`dest`	An array of real numbers to process. The array 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 ($180^\circ/\pi$). The log amplitude is calculated by taking the result of a forward FFT (same as mode 1 above) and forming:

$dB = 10.0\ \mathrm{Log}(power)$ 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()

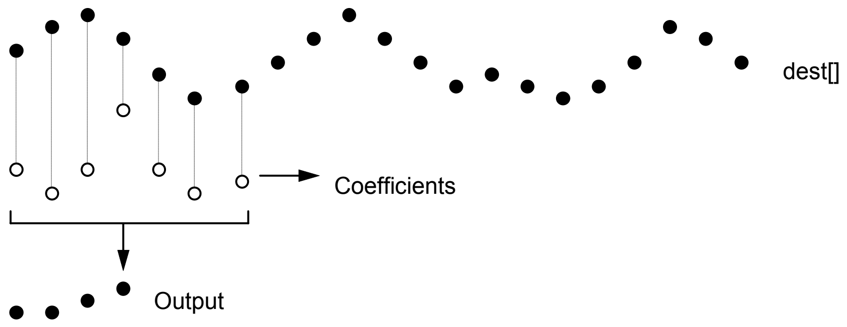
This function filters a real array by replacing each element by the product of a filter array and the surrounding elements of the original array. This implements a FIR (Finite Impulse Response) filter.

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

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

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

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

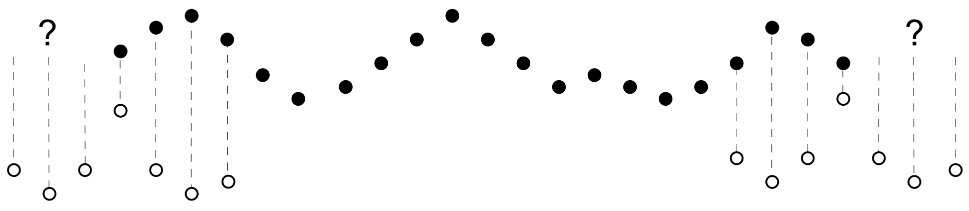


This diagram shows the general principle of the FIR filter. The hollow circles represent the filter coefficients, and the solid circles are the input and output waveforms. Each output point is generated by multiplying the waveform by the coefficients and summing the result. The coefficients are then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

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

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

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



Our 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.
```

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

```
coef[0]:=-1; coef[1]:=1;     'simple difference
ArrFilt(data[], coef[0:2]);  'for differences, equivalent to...
ArrDiff(data[]);             '... just using the differences
```

A simple difference produces a very crude differentiator. A slightly better one is:

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

You can improve the result with more points, for example for 4 points, the coefficients are -0.3, -0.1, 0.1, 0.3 and for five points try -0.2, -0.1, 0.0, 0.1, 0.2. It is more usual to use an odd number of points as this does not cause a shift of the result by half a point. You can use the built-in digital filter support commands such as `FiltCalc()` and `FIRMake()` to generate filter coefficients.

If you use this command to generate 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 just omit the trailing zero coefficients, the output will be time shifted backwards by half the number of coefficients you do supply.

See also: `ArrAdd(), ArrConst(), ArrDiff(), ArrDiv(), ArrDivR(), 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`	An array of real or integer data.

The function is equivalent to the following:

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

See also: `ArrAdd(), ArrConst(), ArrDiff(), ArrDiv(), ArrDivR(), ArrDot(), ArrFFT(), ArrFilt(), 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 reals or integers.
source	An array of reals or integers to multiply the data in `dest`, element by element.
value	A value to multiply the data in `dest`.
Returns	The function returns 0 if all was well, or a negative error code.

If the `dest` array is integer, the multiplication is done as reals and truncated to integer.

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

ArrSub()

This function forms the difference of two arrays or subtracts a constant from an array. Integer overflow is detected with integer destination arrays when the `source` or `value` is a real.

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

`dest`	A real or integer array that holds the result.
`source`	A real or integer array.
`value`	A real or integer value.
Returns	If no overflow is detected, the function returns 0. Overflow is flagged by a negative error code.

The function performs the following operations:

```
var dest[100], source[100], value, i%;
ArrSub(dest[],source[]);          'Is equivalent to...
for i%:=0 to 99 do
    dest[i%] := dest[i%] - source[i%];
    next;

ArrSub(dest[],value);             'Is equivalent to...
for i%:=0 to 99 do
    dest[i%] := dest[i%] - value;
    next;
```

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

ArrSubR()

This function forms the difference of two arrays or subtracts an array from a constant. Integer overflow is detected with integer destination arrays when the `source` or `value` is a real.

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

`dest`	A real or integer array that holds the result.
`source`	A real or integer array.
`value`	A real or integer value.
Returns	If no overflow is detected, the function returns 0. Overflow is flagged by a negative error code.

The function performs the following operations:

```
var dest[100], source[100], value, i%;
ArrSubR(dest[],source[]);         'Is equivalent to...
for i%:=0 to 99 do
    dest[i%] := source[i%] - dest[i%];
    next;

ArrSubR(dest[],value);            'Is equivalent to...
for i%:=0 to 99 do
    dest[i%] := value - dest[i%];
    next;
```

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

ArrSum()

This function forms the sum of the values in an array, and optionally forms the mean and standard deviation.

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

`arr` A real or integer array to process.

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

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

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

Returns The function returns the sum of the array elements

See also:`ArrAdd()`,`ArrConst()`,`ArrDiff()`,`ArrDiv()`,`ArrDivR()`,`ArrDot()`, `ArrFFT()`,`ArrFilt()`,`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 returns the arc tangent of an expression, or the arc tangent of an array:

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

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

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

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

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

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

BinError()

This function is used in a result view with error bins enabled to access the error information. Error bins are enabled for a result view created with `SetAverage()` or with `SetResult()` with 4 added to `flags%`. If you are setting the error information you must set the sweep count with `Sweeps()` first as the sweep count is used to convert the standard deviation into the internal storage format. There are two command variants, the first transfers data for a single bin, the second for an array of bins:

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

`chan%`	The channel number in the 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 a single bin.
`sd[]`	An array used to hold standard deviation values. Values are transferred starting at bin `bin%` in the result view. If the array is too long, extra bins are ignored.
`set%`	If present and non-zero, the values in `sd[]` are copied to the result view. If omitted or zero, values are copied from the result view into `sd[]`.
Returns	The first command variant returns the standard deviation at the time of the call. The second variant returns the number of bins copied. If there are 0 or 1 sweeps of data or errors are not enabled, the result is 0.

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

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

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

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

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

Waveform	The sampling interval between points on the channel.
WaveMark	The sampling interval between points on the channel.
Others/none	The underlying time resolution of the file in seconds.

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.

To find the total range of the x axis in a result view use `BinToX(0)` for the left hand end and `BinToX(MaxTime())` for the right hand end. Do not confuse this with `XLow()` and `XHigh()`, which return the visible range of the x axis in the current view.

In a time view, this is in the underlying time units (set in the microseconds per unit time field). If this value is beyond the range of the x axis, 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()`, `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()`, `BSeek()`, `BWrite()`

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.

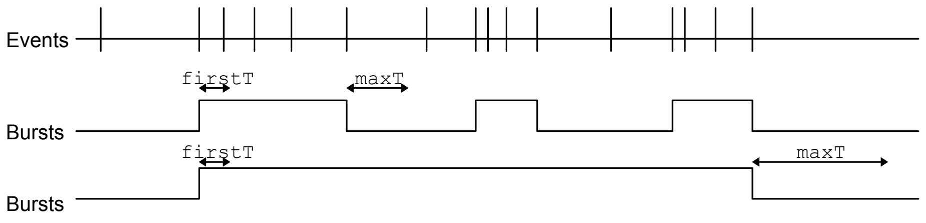
`rel%` This determines to what the new position is relative.
0 Relative to the start of the file (same as omitting the argument)
1 Relative to the current position in the file
2 Relative to the end of the file

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).

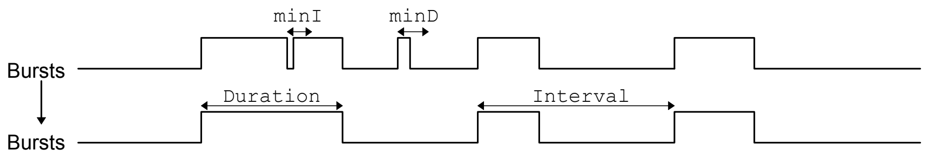
```
Func BurstRevise(mChan%, eChan%, sTime, eTime, minI, minD);
```

`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 = $(\texttt{sdD/meanD})^2 + (\texttt{sdI/meanI})^2$

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 and plus an extra zero byte to mark the end. Use `BWriteSize()` to write a fixed number of bytes.
Returns	It returns the number of arguments for which complete data was written. If an error occurred during the write, a negative code is returned.

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

BWriteSize()

This function writes a variables or arrays as binary into a file opened by `FileOpen()` with a `type%` code of 9. The function can write 8, 16 and 32-bit integers, 32 and 64-bit reals and strings. This allows you to write formats other than the 32-bit integer and 64-bit real used internally by 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()`, `BWrite()`, `BRead()`, `BReadSize()`

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()

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

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

`chan%` A channel in the time or result view.

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

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

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

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

ChanComment$()

This returns or sets the comment string for a channel in a time view. It returns an empty string in a result view. The comment can be up to 71 characters long. It is an error to use this in any other view type.

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

`chan%`	A channel in the time view.
`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. Setting a comment for a result view does nothing, but is not an error.

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 function deletes a channel from a time view or an 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 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 channel deletion if the channel is part of a time view data file. You cannot delete channels that are in the process of 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 function duplicates a time or result view channel. The channel is labelled on screen with the channel number of the "real" channel plus a lower case letter. The lowest free letter is used (from "a"). Use `DupChan()` to find duplicates of a channel

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

`chan%`	The channel number to duplicate; this channel must exist.
Returns	The channel number of the duplicate. Any error stops the script.

The new channel is not displayed. Use `ChanShow()` to make it visible. The following example duplicates a channel and makes it visible:

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

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

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()`

ChanKind()

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

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

`chan%` The channel number. In a result view, channels start at 1, but we accept 0 as meaning the first channel to be compatible with version 3.

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. If you used negative return values to detect them, you must rewrite your code or use `ViewKind()` to be compatible with all versions of Spike2.

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

ChanList()

This function generates a channel list in an array for a time view, result or an XY view. The channels can be filtered to show only a subset of the available channels.

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

`list%`	An integer array to fill with channel numbers. Element 0 is set to the number of channels returned. The remaining elements are channel numbers. If the array is too short, enough channels are returned to fill the array. It is unnecessary to list all the channel numbers for an XY view, since they are numbered contiguously.
`types%`	This argument specifies which channels to return. If omitted, all channels are returned. The values are the same as those defined for the `DlgChan()` function. This argument is ignored in an XY view where all channels are the same type.
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. In a result view, channels start at 1, but we accept 0 as meaning the first channel to be compatible with version 3.

`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 Absolute maximum
2 Mean	6 Curve area	10 Peak to Peak	14 Peak
3 Slope	7 Modulus	11 RMS Amplitude	15 Trough
4 Sum	8 Maximum	12 Standard deviation	

`sPos` The start position for the measurement in x axis units. If `sPos` is greater than or equal to `ePos`, the result is 0. `sPos` is converted to a bin number in result views; the conversion is equivalent to `Trunc(XToBin(sPos))`.

`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()`

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.

```
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. The only types of channel worth creating at version 4.04 are waveform and RealWave as you can use `ChanWriteWave()` to write data. We will add further writing routines in later releases. Codes are:

1 Waveform	4 Level (Event+-)	7 RealMark
2 Event (Event-)	5 Marker (default)	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.

Channels created in this way are given default titles, units and comments. You can set these with the `ChanTitle$()`, `ChanUnits$()`, `ChanComment$()`, `ChanScale()` and `ChanOffset()` routines. 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()`, `ChanWriteWave()`, `FileNew()`, `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, and 1 shows it. Other values are reserved (and currently have the same effect as 1).
Returns	The channel number display state at the time of the call.

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

ChanOffset()

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. In a result view, channels start at 1, but we accept 0 as meaning the first channel to be compatible with version 3.
`offset`	If present, this sets the new channel offset. There are no limits on the value.
Returns	The channel offset at the time of the call if this is a waveform or WaveMark channel, or 0 if it is not.

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

ChanOrder()

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

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

`dest%` The destination channel number.

`pos%` The position to drop the moved channels relative to the destination channel. Use –1 to drop before, 0 to drop on top and 1 to drop after. If you drop a channel between grouped channels, then 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.

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

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

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

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 on which the channel was sampled, or -1 if the view is not a result view, or if the channel is a created channel and was not sampled. Calling this for an event channel sampled on event port 0 returns 0. Likewise, calling this for a waveform channel sampled on ADC port 0 returns 0.

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

ChanProcessAdd()

This adds a channel process to a channel in a time view, matching the effect of the Add button in the Channel Process dialog. Channel processing was added at version 4.04.

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

`chan%` The channel number in the current time view. The first implementation allows processes to be added to waveform and RealWave channels.

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

0 Rectify. Positive values are unchanged, negative values are negated. This operation may cause waveform data with a negative offset to 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.

`arg#` Optional arguments for the process. Each process has a default set of arguments. The arguments correspond to the arguments listed in the Channel Process dialog for each process type. It is an error to provide too many arguments.

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. If you omit `n%`, the function returns the number of arguments for this process. If you supply `n%`, the function returns the argument value at the time of the call.

`arg` An optional argument that changes process argument `n%`.

Returns If `n%` is omitted this returns the number of arguments this process expects. 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 a single process from a channel, all processing from a channel, or all processing 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 this to -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()`

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 (not used yet). The first argument is number 1.

Returns The information requested or a negative error code.

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

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

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

`chan%` The channel number. In a result view, channels start at 1, but we accept 0 as meaning the first channel to be compatible with version 3.

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

Searches are done using the drawing mode set for the channel. 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 to draw the channel in the desired mode, or even for it to be visible; you only need to set the display mode before searching.

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

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

`sT,eT` The search start and end times. If `eT` is less than `sT`, the search is backwards.

`sp1` This is 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 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 for all slope measurements. If omitted, the value 0.0 is used. Set it to 0 if `width` is not required for the mode.

`flag%` This is the sum of flag values and is 0 if omitted. At the moment, the only 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.

Returns A positive time if the search succeeds or -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.

From version 4.08: setting the title of a duplicate channel does not change the title of the original. 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 channel 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.

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

`chan%`	A channel in the time view. In a result view, channels start at 1, but we accept 0 as meaning the first channel to be compatible with version 3. 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%, pos {,&data%{,mode%{,binsz{,trig%|edge%}}}})
```

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

`pos` 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 in 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 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. 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 `pos`.
2	Lines mode, result is the same as mode 1.
3	Waveform mode, this is the only mode for waveform channels.
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.

`binSz` This sets the width of the rate histogram bins and the smoothing period for mean frequency mode when specifying your own mode.

`trig%` The trigger channel for raster displays, we assume raster displays of level data are never required.

`edge%` For level data event channels. This sets which edges of the signal to use for mean frequency, instantaneous frequency and rate modes: 0=both edges, 1=rising edges, 2=falling edges. If `edge%` is omitted, both edges are used.

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 `pos`, and `pos` beyond range of x axis. If `data%` is present, errors cause it to be set to 0.

See also:`ChanData()`

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 function sets the relative vertical space to give a channel or a list of channels. The standard vertical space corresponds to a weight of 1. When Spike2 allocates vertical space, channels are of two types: channels with a y axis and channels without a y axis. Spike2 calculates how much space to give each channel type assuming all channels have a weight of 1. Then the actual space allocated is proportional to the standard space multiplied by the weight factor. This means that if you increase the weight of one channel, all other channels get less space in proportion to their original space.

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

`cSpc`	The specification for the list of channels to process. See the *Script language syntax* chapter for a definition of channel specifiers.
`new`	If present, a value between 0.001 and 1000.0 that sets the weight for all the channels in the list. Values outside this range are limited to the range.
Returns	The command returns the channel weight of the first channel in the list.

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

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 previously written data are ignored. The function was added to Spike2 at version 4.04.

```
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 the colours of items. Colours are set in terms of the colour palette, not directly in terms of colours. XY channels are coloured using `XYColour()`. The colours set for time and result view drawing modes can be overridden by `ChanColour()`.

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

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

1 Time background	10 Rate outline	27 Channel number
2 Waveform channel	11 Rate fill	28 Cursors
3 Events as dots	12 Result background	29 Controls (not used)
4 Events as lines	13 Result lines	30 Grid colour
5 Level events	14 Result dots	31 X and Y Axes
6 Marker data	15 Result skyline	32 XY background
7 Mean frequency	16 Result histogram	33 Not saving to disk
8 Inst. Frequency	17 Result histogram fill	
9 Raster dots	18-26 WaveMark codes 0-8	

`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()`

Conditioner commands

The `Cond…` family of commands control external signal conditioners through the serial ports. These commands support the CED 1902 programmable signal conditioner and the Axon Instruments CyberAmp. Other conditioners may be added in the future.

These commands do not control the serial port used by the conditioner or the type of conditioner supported. When you install Spike2 you must choose the conditioner type and set the serial port.

All these commands require a `port%` argument. This is the physical waveform input port number that the conditioner is attached to. It is not the channel number in 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()`, `CondGain()`, `CondGainList()`, `CondGet()`, `CondOffset()`, `CondOffsetLimit()`, `CondRevision$()`, `CondSet()`, `CondSourceList()`, `CondType()`

CondFilter()

This sets or gets the frequency of 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});
```

`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 cut-off 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.
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()`, `CondGain()`, `CondGainList()`, `CondGet()`, `CondOffset()`, `CondOffsetLimit()`, `CondRevision$()`, `CondSet()`, `CondSourceList()`, `CondType()`

CondFilterList()

This function gets a list of the possible filter frequencies of the conditioner. See the `CondSet()` command for more details of conditioner operation.

```
Func CondFilterList(port%, high%, freq[]);
```

`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. A value of 0 means no filtering of the type selected.
Returns	The number of filtering frequencies of the conditioner or a negative error code.

See also:`CondFilter()`, `CondGain()`, `CondGainList()`, `CondGet()`, `CondOffset()`, `CondOffsetLimit()`, `CondRevision$()`, `CondSet()`, `CondSourceList()`, `CondType()`

CondGain()

This sets and gets the gain of the signal passing through the signal conditioner See the `CondSet()` command for more details of conditioner operation.

```
Func CondGain(port% {,gain});
```

`port%`	The waveform port number that the conditioner is connected to.
`gain`	If present this sets the ratio of output signal to the input signal. If this argument is omitted, the current gain is returned. The conditioner will set the nearest gain it can to the requested value.
Returns	the gain at the time of call, or a negative error code.

See also:`CondFilter()`, `CondFilterList()`, `CondGainList()`, `CondGet()`, `CondOffset()`, `CondOffsetLimit()`, `CondSet()`

CondGainList()

This function gets a list of the possible gains of the conditioner for the selected signal source. See the `CondSet()` command for more details of conditioner operation.

```
Func CondGainList(port%, gain[]);
```

`port%`	The waveform port number that the conditioner is connected to.
`gain[]`	An array of reals holding the conditioner gains for the selected signal source. If a conditioner (for example, 1902) has a fixed set of gains, this is the set of gain values. If the conditioner supports continuously variable gain, the first two elements of this array hold the minimum and the maximum values of the gain.
Returns	The number of gain values if the conditioner has a fixed set of gains or 2 if the conditioner has continuously variable gain. In the case of an error, a negative error code is returned.

See also:`CondFilter()`, `CondFilterList()`, `CondGain()`, `CondOffset()`, `CondOffsetLimit()`, `CondSet()`, `CondSourceList()`

CondGet()

This function gets the input signal source of the signal conditioner, and the conditioner settings for gain, offset, filters and coupling. The settings are returned in arguments which must all be variables. See the `CondSet()` command for more details of conditioner operation.

```
Func CondGet(port%, &in%, &gain, &offs, &low, &hi, &notch%, &ac%);
```

`port%`	The waveform port number that the conditioner is connected to.
`in%`	Returned as a zero-based index number of the input signal source of the conditioner.
`gain`	Returned as the ratio of output signal amplitude to the input signal amplitude (ignoring effects due to 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.
Returns	0 if all well or a negative error code.

See also: `CondFilter()`, `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: `CondFilter()`, `CondFilterList()`, `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. See the `CondSet()` command for more details of conditioner operation.

```
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: `CondFilter()`, `CondFilterList()`, `CondGain()`, `CondGainList()`, `CondGet()`, `CondOffset()`, `CondRevision$()`, `CondSet()`, `CondSourceList()`, `CondType()`

CondRevision$()

This function returns the name and version of the signal conditioner as a string or a blank string if there is no conditioner for the port.

```
Func CondRevision$(port%);
```

`port%`	The waveform port number that the conditioner is connected to.
Returns	This returns a string describing the signal conditioner. The strings defined so far are: “`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. If there is no conditioner attached to the port it returns an empty string.

See also: `CondFilter()`, `CondFilterList()`, `CondGain()`, `CondGainList()`, `CondGet()`, `CondOffset()`, `CondOffsetLimit()`, `CondSet()`, `CondSourceList()`, `CondType()`

CondSet()

This sets the input signal source, gain, offset, filters and coupling of the conditioner. All values are requests. The actual values set will depend on the capabilities of the conditioner. In all cases, the command sets the nearest value to the 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%});
```

`port%` The waveform port number that the conditioner is connected to.

`in%` This is a zero-based index of the input signal source. A conditioner can have several different signal sources, for example, the 1902 Mk III supports `Grounded`, `Single ended`, `Normal Diff`, `Inverted Diff`, etc. Different conditioners of the same type may have different sources. `CondSourceList()` returns the whole list of the possible signal sources of your conditioner. You select a signal source by setting `in%` to its index number in the list.

`gain` This is the desired ratio of output signal amplitude 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 cut-off 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 a value of 0, there is no low-pass filtering. The actual filter value set depends on the capability of the signal conditioner.

`high` If present and greater than 0, it is a cut-off frequency of the high-pass filter. High-pass filters are used to reduce the low-frequency content of the signal. This frequency 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 too. If `ac%` is present with a value greater than 0, the signal conditioner is AC coupled. If omitted or 0, the signal conditioner is DC coupled.

Returns 0 if all well or a negative error code.

See also: `CondFilter()`, `CondFilterList()`, `CondGain()`, `CondGainList()`, `CondGet()`, `CondOffset()`, `CondOffsetLimit()`, `CondRevision$()`, `CondSourceList()`, `CondType()`

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. See the `CondSet()` command for more details of conditioner operation.

```
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$[]` is an array of strings, as many sources as will fit in the string array are returned.
Returns	If `in%` is greater than or equal to 0, it returns 1 or a negative error code. If `in%` is omitted, it returns the number of signal sources or a negative error code.
See also:	`CondFilter()`, `CondFilterList()`, `CondGain()`, `CondGainList()`, `CondGet()`, `CondOffset()`, `CondOffsetLimit()`, `CondRevision$()`, `CondSet()`, `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:	`CondFilter()`, `CondFilterList()`, `CondGain()`, `CondGainList()`, `CondGet()`, `CondOffset()`, `CondOffsetLimit()`, `CondRevision$()`, `CondSet()`, `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()`, `Exp()`, `Frac()`, `Ln()`, `Log()`, `Max()`, `Min()`, `Pow()`, `Rand()`, `Round()`, `Sin()`, `Sqrt()`, `Tan()`, `Trunc()`

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 function returns the position of a cursor and optionally sets a new position. The position units are seconds in a time view, bin number in a result view and in x axis units in an XY view. 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 position of the cursor. If the new position is out of range of the x axis, it is limited to the x axis. In a time view the position is a time, 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 or -1 if the cursor doesn't exist.

Examples:

```
Cursor(1,20);             'Set cursor 1 at position 20
where := Cursor(1);       'Get cursor position
```

See also:`BinToX()`, `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$|min, end$
                          {,def$ {,sp1 {,sp2 {,width {,ref$}}}}});
```

`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	
6 Threshold	13 Slope trough	

`ch%` This is the time view channel number to use for searches. Set this to 0 for modes that do not require a channel.

`str$` This is an expression that sets the start time for the search for most search modes. In expression mode (16) this is the expression to evaluate.

`min` This argument is used in place of `str$` for cursor 0 in non-expression mode. It sets the minimum step.

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

`def$` If the cursor seek operation fails, and this string is present and evaluates to a valid time, the cursor is positioned at the time and the cursor position is valid.

`sp1` This is 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 if `sp1` is not required for the mode.

`sp2` This is hysteresis for threshold crossings and percent for percent repolarisation or percentage slope. If omitted, the value 0.0 is used. Set it to 0 if `sp2` is not required for the mode.

`width` This is the width in seconds for all slope measurements and also for the reference level measurement in percent repolarisation mode. If omitted, the value 0.0 is used. Set it to 0 if `width` is not required for the mode.

`ref$` This is an expression used for repolarisation percentage mode, where it is the 100% time, in seconds.

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()`, `CursorNew()`, `CursorSearch()`, `CursorValid()`, `MeasureChan()`, `MeasureX()`

CursorActiveGet()

This function retrieves active cursor parameters set by `CursorActive()` or by 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$`	5 `sp1`	7 `width`	9 `ref$`
2 `str$`	4 `def$`	6 `sp2`	8 `min`	

`val$` This is updated with the parameter value for `str$`, `end$`, `def$` and `ref$` when `item%` is 2, 3, 4 or 9. It is not required for the other values of `item%`.

Returns The function result is the parameter value for numeric parameters `ch%`, `sp1`, `sp2`, `width` and `min`. Otherwise it returns the active cursor mode.

See also:`CursorActive()`,`CursorSearch()`,`CursorValid()`,`MeasureChan()`, `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 command sets (or gets) the cursor label style for the current view. Cursors can be annotated with a position and/or the cursor number, or with a user-defined string:

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

`style%` Label styles are: 0=None, 1=Position, 2=Number, 3=Both, 4=User-defined. Unknown styles cause no change. Style 4 is used with a format string.

`num%` Used with style 4 only. A value of -1 or omitting the argument selects all cursors, 0-9 selects one cursor. **In version 3, <=0 selected all cursors.**

`form$` Cursor 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 The cursor 1 style before any change. If `style%` is omitted, the current 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 new 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()`, `CursorActive()`, `CursorDelete()`, `CursorLabel()`, `CursorRenumber()`, `CursorSet()`, `CursorVisible()`, `XToBin()`

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.

```
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 "/". **Warning**: This command does not exist in version 2. Default arguments 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 has two functions. It can open the debug window so you can step through your script, set breakpoints and display and edit variables. From version 3 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()`

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.

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

`list%` This determines what to list. A value of 0 lists a summary of the options, 1 lists fixed objects (constants and operators), 2 lists permanent objects (constants, operators and built in commands). Values greater than 2 list information for the object with that number.

`opt%` This optional argument (default value 0) sets the additional object information to list. 1= list the index number, 2= list the type, 3= list both index and type.

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$()`, `UCase$()`, `Val()`

Discriminator (CED 1401-18) support

The `Discrim…` family of commands supports the 1401-18 event discriminator card, which is available for the standard 1401 and the *1401plus* only. You can also control the 1401-18 card interactively; see the Sample menu.

If you use these commands with the interactive discriminator dialog active, the dialog changes to show any changes made from the script. You cannot change the current channel the dialog displays (except with `DiscrimClear()`), so you will only see changes in the dialog if the channel is the same as the current channel in the dialog.

See also:`DiscrimChanGet()`, `DiscrimChanSet()`, `DiscrimClear()`, `DiscrimLevel()`, `DiscrimMode()`, `DiscrimMonitor()`, `DiscrimTimeOut()`

DiscrimChanGet()

This gets the input, spike2 event port, 1401 event input E1 or E3, mode, lower and higher trigger levels, and the time-out for modes 7 and 8 of a discriminator channel.

```
Func DiscrimChanGet(chan%, &in%, &out%, &mode%, &low, &hi, &tOut);
```

`chan%` the discriminator channel number (0-7).

`in%` Returned holding the source of the discriminator channel as:

0	The discriminator is not used
1	From the 1401 front panel digital input port
2	From the front panel event input (channels 0-4) or ADC Ext (channel 5)

`out%` Returned holding a code that describes how the discriminator output, event inputs and digital inputs are connected to the Spike2 event ports and the E1 (digital marker trigger) and E3 (start sampling trigger). The E1 and E3 values are only valid for channels 1 and 3.

	Spike2 event port	**E1 (digital marker) or E3 (start sampling)**
0	Disconnected.	Disconnected
1	1401 digital input	Disconnected
2	Discriminator output	Disconnected
4	Disconnected	To front panel E1 or E3
5	1401 digital input	To E1 or E3
6	Discriminator output	To E1 or E3
8	Disconnected	Discriminator output
9	1401 digital input	Discriminator output
10	Discriminator output	Discriminator output

`mode%` the mode of the discriminator channel is returned in this variable:

1	detect a level below a threshold
2	detect a level above a threshold
3	pulse on rising signal through a threshold
4	pulse on falling signal through a threshold
5	detect a level inside a region set by two thresholds
6	detect a level outside region set by two thresholds
7	pulse when signal returns below lower threshold, without going above the higher threshold, within a programmable time out period
8	pulse when signal returns above higher threshold, without going below the lower threshold, within a programmable time out period

`low` Returned holding the lower threshold level in Volts.

`high` Returned holding the higher threshold level in Volts.

`tOut` Returns the programmable time out period used in modes 7 and 8.

Returns 0 if all well or a negative error code.

See also:`DiscrimChanSet()`, `DiscrimClear()`, `DiscrimLevel()`, `DiscrimMode()`, `DiscrimMonitor()`, `DiscrimTimeOut()`

DiscrimChanSet()

This sets the input, output, mode, lower and higher trigger levels, and the time-out (if the mode is 7 or 8) of the discriminator channel.

```
Func DiscrimChanSet(chan%, in%, out%{, mode%, low, high, tOut});
```

`chan%` the discriminator channel number (0-7).

`in%` Sets the source of the discriminator channel as:

0 Disconnected
1 From the 1401 front panel digital input port
2 From the front panel event input (channels 0-4) or ADC Ext (channel 5). For discriminator channels 6 and 7, you cannot set `in%` to 2 as there is no 1401 event input available as the source.

`out%` Sets how the discriminator output, event inputs and digital inputs connect to the Spike2 event ports and the E1 (digital marker) and E3 (start sampling) triggers. If `in%` is 0, `out%` is forced to 1. If both `in%` and `out%` are 1, `out%` is forced to 2. Possible `out%` values are:

0 Spike2 event port disconnected
1 Spike2 event port connected to 1401 digital input
2 Spike2 event port connected to discriminator output

When the channel is 1 or 3 you can choose what the 1401 event input E1 (digital marker) or E3 (start sampling) trigger connects to by adding 0, 4 or 8 to `out%`. If `in%` is 0, adding 4 is forced. If `in%` is 2, adding 4 is treated as adding 8.

0 E1 or E3 is disconnected
4 E1 or E3 is connected to the front panel E1 or E3
8 E1 or E3 is connected to the discriminator output

`mode%` The discriminator channel mode (1-8). If omitted, no change is made.

1 detect a level below a threshold
2 detect a level above a threshold
3 pulse on rising signal through a threshold
4 pulse on falling signal through a threshold
5 detect a level inside a region set by two thresholds
6 detect a level outside region set by two thresholds
7 pulse when signal returns below lower threshold, without going above the higher threshold, within a programmable time out period
8 pulse when signal returns above higher threshold, without going below the lower threshold, within a programmable time out period

`low` Sets the lower threshold level in volts between -5 Volts and the higher threshold level value. If omitted, no change is made. If the lower threshold level is given greater than the higher one, it will be set equal to the higher level.

`high` Sets the higher threshold level in Volts between the lower threshold level value and +5 Volts. If omitted, no change is made.

`tOut` Sets the time out period in seconds (0.00002 - 0.65535). This is only meaningful in mode 7 or 8. If omitted, no change is made.

Spike2 stores and uses the values for the threshold levels and time out period in terms of the resolution of the 1401-18 card. To find out the real value of the threshold levels and time out, use `DiscrimLevel()` and `DiscrimTimeOut()`. The difference is usually very small.

Returns 0 if the operation completed without a problem, or a negative error code.

See also: `DiscrimChanGet()`, `DiscrimClear()`, `DiscrimLevel()`, `DiscrimMode()`, `DiscrimMonitor()`, `DiscrimTimeOut()`

DiscrimClear()

This sets the Discriminator configuration dialogue contents to a standard state. All the discriminator channels are disconnected. Spike2 event ports are connected to digital inputs. E1 (digital marker trigger) and E3 (start sampling trigger) are connected to the front panel E1 and E3. All the discriminator channels have mode 3, and the lower and higher trigger levels are set to 1.25 and 2.5 Volts. The ADC monitor channel is set to 15. If the discriminator dialog is open, channel 0 becomes the current channel.

```
Proc DiscrimClear();
```

See also: `DiscrimChanGet()`, `DiscrimChanSet()`, `DiscrimLevel()`, `DiscrimMode()`, `DiscrimMonitor()`, `DiscrimTimeOut()`

DiscrimLevel()

This sets or gets the lower or higher threshold level values of the discriminator channel.

```
Func DiscrimLevel(chan%, which%, {level});
```

`chan%` The discriminator channel number (0-7).

`which%` Selects which level to set or get: 0 = lower, 1 = upper threshold level.

level If present it sets the threshold value in Volts:

If `which%` is 0, it is the lower level. It should lie between -5 volts and the higher level. If it is set greater than the higher level, it is set equal to the higher one.

If `which%` is 1, it is the higher level. It should lie between the lower level and 5 Volts. If it is set less than the lower level, it is set equal to the lower one

Returns the lower or higher threshold level of the discriminator channel at the time of call, or a negative error code.

See also: `DiscrimChanGet()`, `DiscrimChanSet()`, `DiscrimClear()`, `DiscrimMode()`, `DiscrimMonitor()`, `DiscrimTimeOut()`

DiscrimMode()

This sets or gets the mode of the discriminator channel.

```
Func DiscrimMode(chan%, {mode%});
```

`chan%` The discriminator channel number (0-7).

`mode%` The mode of the discriminator channel (1-8). If omitted, the current mode of the discriminator channel is returned.

Returns The discriminator channel mode at the time of call, or a negative error code.

1	detect a level below a threshold
2	detect a level above a threshold
3	pulse on rising signal through a threshold
4	pulse on falling signal through a threshold
5	detect a level inside a region set by two thresholds
6	detect a level outside region set by two thresholds
7	pulse when signal returns below lower threshold, without going above the higher threshold, within a programmable time out period
8	pulse when signal returns above higher threshold, without going below the lower threshold, within a programmable time out period

See also: `DiscrimChanGet()`, `DiscrimChanSet()`, `DiscrimClear()`, `DiscrimLevel()`, `DiscrimMonitor()`, `DiscrimTimeOut()`

DiscrimMonitor()

This sets or gets the current ADC monitor channel in the Discriminator configuration dialog.

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

`chan%`	Sets the ADC monitor channel number (0-15). If this is omitted, the current ADC monitor channel number is returned.
Returns	the ADC monitor channel number at the time of call, or a negative error code.
See also:	`DiscrimChanGet()`, `DiscrimChanSet()`, `DiscrimClear()`, `DiscrimLevel()`, `DiscrimMode()`, `DiscrimTimeOut()`

DiscrimTimeOut()

This sets or gets the time out period of mode 7 or 8 for the discriminator channel.

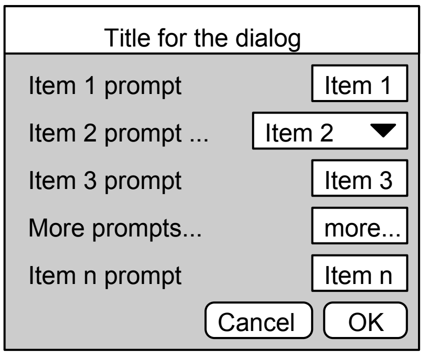
```
Func DiscrimTimeOut(chan%, {tOut});
```

`chan%`	the discriminator channel number (0-7).
`tOut`	the programmable time out period in seconds (0.00002 - 0.65535). This is only meaningful in mode 7 or 8. If this is omitted, the current time out period is returned.
Returns	the time out period at the time of call, or a negative error code.
See also:	`DiscrimChanGet()`, `DiscrimChanSet()`, `DiscrimClear()`, `DiscrimLevel()`, `DiscrimMode()`, `DiscrimMonitor()`

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 exact appearance of the dialog depends on the system. The dialog is arranged in terms of items. Unless you specifically request otherwise, the dialog items are stacked vertically above each other.



The dialog has a title of your choosing at the top. There are OK and Cancel buttons at the bottom of the dialog. When the dialog is used, pressing the Enter key is equivalent to clicking on OK.

This form of dialog is very easy to program; there is no need to specify any position or size information, the system works it all out. Some users require more complicated dialogs, with control over the field positions. This is also possible, but more complicated to program. You are allowed up to 256 fields in a dialog.

In the more complex case, you specify the position (and usually the width) of the box used for user input. This allows you to arrange data items in the dialog in any way you choose. It requires more work as you must calculate the positions of all the items.

Dialog units

Positions within a dialog are set in *dialog units*. In the x (horizontal) direction, these are in multiples of the maximum width of the characters ‘0’ to ‘9’. In the y (vertical) direction, these are in multiples of the line spacing used for simple dialogs. Unless you intend to produce complex dialogs with user-defined positions, you need not be concerned with dialog units at all.

Dialog example

The example dialog shown above could be created by this code:

```
var ok%, item1%, item2%, item3, item4$, item5;
DlgCreate ("Title for the dialog");      'start new dialog
DlgInteger(1,"item 1 prompt",0,10);      'integer, range 0-10
DlgChan   (2,"item 2 prompt",1);         'Waveform channel list
DlgReal   (3,"item 3 prompt",1.0,5.0);   'real, range 1.0-5.0
DlgString (4,"More prompts...",6);       'string, any characters
DlgReal   (5,"item n prompt",-10.0,0.0); 'real, range -10-0
item4$ := "more...";
ok% := DlgShow(item1%, item2%, item3, item4$, item5); 'show dialog
```

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

DlgChan()

You often need to select a channel of a particular type from a time view. This function defines a dialog entry that lists channels that meet a specification. For simple dialogs, the `wide`, `x` and `y` arguments are not used. Channel lists are checked or created when the `DlgShow()` function runs. If the current view is not a time view, the list will be 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 text to display as a prompt.

`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 is an integer code that determines the channels to be displayed. You can select channels of particular types by adding together the following codes (the numbers in round brackets are the same codes in hexadecimal format):

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

If none of the above values are used, then the list includes all channels. The following codes can be added to exclude channels from the list created above:

1024	Exclude visible channels (0x400)
2048	Exclude hidden channels (0x800)
4096	Exclude disk channels but not duplicates of them (0x1000)
8192	Exclude memory channels but not duplicates of them (0x2000)
16384	Exclude duplicates of disk or memory channels (0x4000)
32768	Exclude selected channels (0x8000)
65536	Exclude non-selected channels (0x10000)

Finally, the following codes allow special entries to be added to the list:

131072	Add `None` as an entry in the list, returns 0 (0x20000)
262144	Add `All channels` as an entry in the list, returns -1 (0x40000)
524288	Add `All visible channels` as an entry, returns -2 (0x80000)
1048576	Add `Selected` as an entry in the list, returns -3 (0x100000)

`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.
`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, 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()`

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 text to display to the right of the check box.
`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).

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

DlgCreate()

This function starts the definition of a dialog. It also kills off any previous dialog that might be partially defined. For simple dialogs, the optional arguments are not used.

```
Func DlgCreate(title${, x{, y{, wide{, high{, help%|help$}}}}});
```

`title$`	A string holding the title for the dialog.
`x,y`	Optional, taken as 0 if omitted. The position of the top left hand corner of the dialog. The positions are in percentages of the screen size. The value 0 means centre the dialog. Values out of the range 0 to 95 are limited to the range 0 to 95.
`wide`	The width of the dialog in dialog units. If this is omitted, or set to 0, 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`	This is a string or numeric identifier that identifies the help page to be displayed if the user requests help when the dialog is displayed.
Returns	This function returns 0 if all was well, or a negative error code.

For simple use, only the first argument is needed. The remainder are for use with more complicated menus where precise control over menu items is required.

Use of & in prompts In the functions that set an item with a prompt, if you precede a character in the prompt with an ampersand &, the following character is used by Windows as a short-cut key to move to the field and the character is underlined. Ampersand characters are ignored on systems that do not use this mechanism (for example, the Macintosh).

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

DlgInteger()

This function defines a dialog entry that edits an integer. The numbers you enter may not contain a decimal point. For simple dialogs, the `wide`, `x` and `y` arguments are not used.

```
Proc DlgInteger(item%, text$|wide, lo%, hi%{, x{, y}});
```

`item%`	This sets the item number in the dialog in the range 1 to the number of items.
`text$`	The text to display as a prompt. Text is displayed left justified in the dialog box.
`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, this takes the value of `item%`. It is the position of the bottom of the number entry box in dialog units.

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()`, `DlgCreate()`, `DlgLabel()`, `DlgList()`, `DlgReal()`, `DlgShow()`, `DlgString()`, `DlgText()`

DlgLabel()

This function sets an item with no editable part that is used as a label. For simple dialogs, the `wide`, `x` and `y` arguments are not used. You can add text to a dialog without using an item number with `DlgText()`.

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

`item%`	This sets the item number in the dialog in the range 1 to the number of items.
`text$`	The text to display.
`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, 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:`DlgChan()`, `DlgCheck()`, `DlgCreate()`, `DlgInteger()`, `DlgList()`, `DlgReal()`, `DlgShow()`, `DlgString()`, `DlgText()`

DlgList()

This defines a dialog item for a one of n selection. Each of the possible items to select is identified by a string. For simple dialogs, the `wide`, `x` and `y` arguments are not used.

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

`item%`	This sets the item number in the dialog in the range 1 to the number of items.
`text$`	The text to display as a prompt.
`wide`	This is an alternative to the prompt. It sets the width of the box in which the user selects an item. If the width is not given the number entry box has a default width of the longest string in the list or 20, whichever is the smaller.
`list$`	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 2.
`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:

```
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()`

DlgReal()

This function defines a dialog entry that edits a real number. For simple dialogs, the `wide`, `x` and `y` arguments are not used.

```
Proc DlgReal(item%, text$|wide, lo, hi{, x{, y}});
```

`item%`	This sets the item number in the dialog in the range 1 to the number of items.
`text$`	The text to display as a prompt. Prompts are left justified in the dialog.
`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.

The variable passed into `DlgShow()` should be a real number. The field will start with the value of the variable if it is in the range, otherwise the value is limited to `lo` or `hi`.

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

DlgShow()

This function displays the dialog you have built and returns values from the fields identified by item numbers, or makes no changes if the user kills the dialog with the Cancel button. Once the dialog has 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 on the Cancel button, or 1 if the user clicked on OK.

For each dialog item with an item number, 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. Items created with `DlgLabel()` must have a variable too, even though it is not changed.

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 a simple variable, you can pass an array. An array with `n` elements matches `n` items in the dialog. The array type must match the items.

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

DlgString()

This defines a dialog entry that edits a text string. You can limit the characters that you will accept in the string. For simple dialogs, the `wide`, `x` and `y` arguments are not used.

```
Proc DlgString(item%, text$|wide, max%{, legal${, x{, y}}});
```

`item%`	This sets the item number in the dialog in the range 1 to the number of items.
`text$`	The text to display as a prompt.
`wide`	This is an alternative to the prompt. It sets the width in dialog units of the box in which the user types the string. If the width is not given the number entry box has a default width of `max%` or 20, whichever is the smaller.
`max%`	The maximum number of characters allowed in the string.
`legal$`	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, this takes the value of `item%`. It is the position of the bottom of the string entry box in dialog units.

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:`DlgChan()`, `DlgCheck()`, `DlgCreate()`, `DlgInteger()`, `DlgLabel()`, `DlgList()`, `DlgReal()`, `DlgShow()`, `DlgText()`, `Input$()`

DlgText()

This places non-editable text in the dialog box. This is different from `DlgLabel()` as it has no item number and so does not require a variable in the `DlgShow()` function.

```
Proc DlgText(text$, x, y);
```

`text$`	A text string to place in the dialog.
`x,y`	The position of the bottom left hand corner of the first character in the string, in dialog units. Set `x` to 2 for the same label position as `DlgLabel()`.

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

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.

```
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.
`size`	The width of the window in the same units as `from`. A negative `size` is ignored.

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()`

DrawAll()

This routine updates all views with invalid regions.

```
Proc DrawAll();
```

See also:`Draw()`

DrawMode()

This sets and reads the channel display mode in a time or result view. You can set the display mode for hidden channels. Modes -13 to -6 and 9 to 13 do not exist in version 2.

```
Func DrawMode(chan%{,mode%});      Return mode or information (negative mode%)
Func DrawMode(cSpc, mode%{,dotSz%|binSz{,trig%|edge%|opt%|err%}});
Func DrawMode(cSpc, 9{, fftSz%, wnd%, top, range, xInc%, skip%});
```

`chan%` The channel number used to read back information with a negative `mode%`.

`cSpc` The channel specification or -1 to mean all channels, -2 for visible channels and -3 for selected channels. The first channel in a result view is 1, but we allow 0 to mean 1 for backwards compatibility. Setting a draw mode for a bad channel number has no effect.

`mode%` If present and positive, this sets the display mode. If an inappropriate mode is requested, no change is made. If `mode%` is absent, the function returns the mode. 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 draw mode 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 and WaveMark points. No WaveMark codes are displayed (faster than WaveMark mode).
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. `binSz` sets the width of each bin in seconds.
6 Mean frequency mode. `binSz` sets the time period.
7 Instantaneous frequency mode. `dotSz%` 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%` If this is 1, Spike2 uses only one data block for each vertical strip of sonogram. If you have many megabytes of data, this will reduce the calculation time (but only shows samples of the sonogram). This is normally 0. 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 and WaveMark channels.

For a result view channel the modes are listed below.

0	The standard drawing mode for the channel.
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	Reserved for Cubic spline mode. `err%` can be used.

`dotSz%` Sets the dot or tick size to use on screen or the point size to use on a printer. 0 is the smallest size available. The maximum size is 10. Use -1 for no size change.

`binSz` Sets rate histogram bin width and the mean frequency smoothing period.

`trig%` This is the trigger channel for 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.

`opt%` Used in result views only for raster channels. 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 4 to sort by auxiliary variable 0, add 8 to sort by auxiliary variable 1, add 16 to show symbols.

`err%` The drawing style for errors: 0=none, 1=1 SEM, 2=2 SEM, 3=SD.

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%`
-2	`dotSz%`	-5	`edge%`	-8	`top`	-11	`skip%`		
-3	`binSz`	-6	`fftSz%`	-9	`range`	-12	`opt%`		

See also:`Draw()`, `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, this returns the view handle of the duplicate, or 0 if the duplicate does not exist. If `num%` is 0 or omitted, this returns the number of duplicates. The following code illustrates the use of `Dup()`.

```
var maxDup%, i%, dvh%;      'declare variables
maxDup% := Dup(0);          'Get maximum numbered duplicate
for i% := 1 to maxDup% do  'loop round all possible duplicates
   dvh% := Dup(i%);         'get handle of this duplicate
   if (dvh% > 0) then       'does this duplicate exist?
      PrintLog(view(dvh%).WindowTitle$()+"\n");'print window title
   endif;
   next;
```

See also:`App()`, `View()`, `WindowDuplicate()`

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. See `ChanDuplicate()` for more information on duplicate channels.

```
Func DupChan(chan%{, num%});
```

`chan%`	A channel number in the current view. The channel must 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. The index is not related to the character added to a channel number to indicate a duplicate.
Returns	If `num%` is greater than 0, this returns the channel number of the duplicate, or 0 if the duplicate does not exist. If `num%` is 0, this returns the channel number that `chan%` was duplicated from or `chan%` if it is the original. If `num%` is omitted, this returns the number of duplicates.

See also:`ChanDuplicate()`

EditClear()

In a result view, this command zeros the data. 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. The effect depends on the type of the current view. Text windows copy data as text. Time views copy as a bitmap, as a scaleable image and as text in the format set by `ExportTextFormat()`, `ExportChanFormat()` and `ExportChanList()`. Result and XY views can copy as a bitmap, text and as a scaleable image.

```
Func EditCopy({as%});
```

`as%` This sets how to copy data when several formats are possible. If omitted, all formats are used. When only one format is possible, the argument is ignored. Systems that cannot implement some or all of these features ignore the argument. The format value is the sum of:

1	Copy as a bitmap
2	Copy as a scaleable image (Windows metafile, Macintosh PICT)
4	Copy as text

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()`, `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 copied to the clipboard in cases where there are several formats possible. Currently only text may be cut. 1 Copy as a bitmap 2 Copy as a scaleable image (metafile) 4 Copy as text
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()`

EditPaste()

You may only paste into a text window when the clipboard contains text. The contents of the clipboard are inserted into the text at the current caret. If text is already selected, it is replaced by the clipboard contents.

```
Func EditPaste();
```

Returns The function returns the number of characters inserted.

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.

```
Proc Eval(arg);
```

`arg` A real or integer number or a string.

If you use `Eval()` it will suppress any run-time error messages as it uses the same mechanism as the error system. A common use of `Eval()` in a script is to report an error condition during debugging, for example:

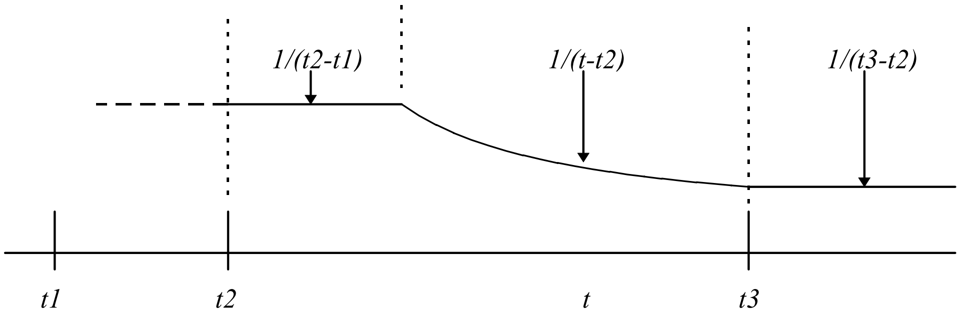
```
if val<0 then Eval("Negative value"); Halt; endif;
```

See also:`Debug()`, `Error$()`, `Print()`, `PrintLog()`

EventToWaveform()

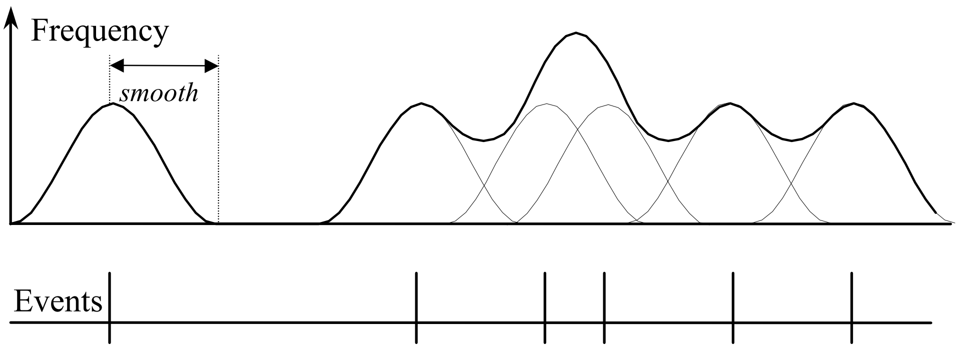
This function creates a new waveform channel from an event channel. The waveform channel contents depend on the frequency of the events. You can choose to convert based on instantaneous frequency, or on a smoothed frequency.

Instantaneous frequency Spike2 calculates instantaneous frequency from the reciprocal of event intervals. To obtain values 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 when there are three events at times *t1*, *t2* and *t3*. The output from *t1* to *t2* depends on what happened before *t1*. From time *t2* to *t2*+(*t2-t1*), the output is the reciprocal of 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 each event by a waveform fragment that has unit area below it. The waveform is symmetrical about the event time. If the event is at time *t*, the waveform fragment extends from time *t-smooth* to *t+smooth*.



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% {,shape[]}});
```

`smooth` The period to smooth the event data over, in seconds. If this is 0 or negative the conversion is instantaneous frequency, otherwise an event at time `t` is spread over a time range `t-smooth` to `t+smooth` seconds. Unless you supply the `shape[]` 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 the range 0.000001 to 1000.0 seconds. Values outside this range are errors. This interval is rounded to the nearest multiple of the microseconds per time unit set for the file. The `BinSize()` command will return the actual sampling interval. Before Spike2 version 4.03, the interval was rounded to the nearest multiple of microseconds per time unit times time units per ADC convert for the file. If you set an interval that could not be achieved by older versions of Spike2, the resulting data file cannot be opened by versions of Spike2 prior to 4.02.
`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.
`shape`	This optional argument is only used when `smooth` is greater than 0. Normally a raised cosine function is used as the smoothing function. If you provide this array argument, it replaces the cosine. You can set any size of array, but we suggest between 50 and 1000 points. The built-in raised cosine function uses 250 points. The code to generate the built-in function is:

```
var raisedCos[250], index%;
const pi := 3.14159;
for index% := 0 to 249 do
    raisedCos[index%] := 1.0 + Cos(index%*pi/250);
    next;
```

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

A common use of this command is to investigate slow frequency changes in a sequence of events. For example, heart rate is modified by respiration. If you have an event channel with each event representing a heart beat (with an expected frequency of around 70 beats per minute, or 1.17 Hz) and you are looking for a variation due to respiration at about 0.24 Hz, you need to convert the heartbeats into a rate signal. The expected result using this command would be a steady frequency of (say) 1.2 Hz, plus a small oscillation around 0.25 Hz. The small oscillation can be detected with `SetPower()`.

If we want a 0.01 Hz resolution, we will need some 100 seconds of data per power spectrum block. If we used a 1024 point transform, we need a sample interval of about 0.1 Hz (so that 1024 points spans just over 100 seconds). The `SetPower()` result will have 512 bins, spanning the range 0 to 5.00 Hz. A smoothing period of some 5-10 seconds would be appropriate.

See also:`BinSize()`, `SetPower()`

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 command sets the channel text export format for use by `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. It is ignored and can be omitted for waveform and event channels. If omitted or set to 0, the channels are treated as their own type. The codes are the same as for the `type%` field.

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()` in a time view. The current view must be a time view. Repeated calls to this routine are additive. Call it with one or no arguments to clear the list.

```
Proc ExportChanList(sTime, eTime, cSpc {,cSpc...});
Proc ExportChanList({flags%});
```

`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.

The following example exports channels 1 and 2 from 90.0 to 100.0 seconds as the first 10 seconds of a new file `fred.smr`.

```
ExportChanList(1);            'Clear export list, sets zero shift
ExportChanList(90, 100, 1, 2);'set channels 1 and 2
FileSaveAs("fred.smr", 0);    'export to new data file
```

See also:`FileSaveAs()`, `ExportChanFormat()`, `ExportTextFormat()`, `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$`	The character to place at the start and end of each text string in the output. If omitted, a double quote character is used.
`sep$`	The character to separate multiple data items on a line. If omitted, tab is set.

See also:`FileSaveAs(),ExportChanList(),ExportChanFormat(),EditCopy()`

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. `View(fh%);FileClose()` is OK.

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 `Spike4\import` folder.

```
Func FileConvert$(src${, dest${, flag%{ ,&err%}}});
```

`src$`	This is the name of the file to convert. The file extension is used to determine the file type (unless `flag%` bit 0 is set). Known file extensions include: `abf`, `cfs`, `cnt`, `cut`, `dat`, `eeg`, `ewb`, `ibw`, `ibw`, `son` and `uff`. We expect to add more.
`dest$`	If this is present, it sets the destination file. If this is not a full path name, the name is relative to the current directory. If you do not supply a file extension then Spike2 appends `".smr"`. If you set any other file extension, Spike2 cannot open the file as a Spike2 data file. If you do not supply this argument, the converted file will be written to the same folder as the source file using the original file name with the file extension changed to `.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 returned as 0 if the file was converted, otherwise it is returned holding a negative error code.
Returns	The full path name of the created file, or an empty string if the file was not converted.

See also:`FileOpen()`, `FilePath$()`, `FilePathSet()`, `FileList()`

FileCopy()

This function copies a source file to a destination file. File names can be specified in full or relative to the current directory. Wildcards cannot be used. (Added at version 4.08).

```
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$()

Since version 4.03, the time and date at which sampling started is saved in the data file. This function returns a string holding the date. Use `FileTimeDate()` to get the date as numbers. The current view must be a time view. The arguments are exactly the same as for the `Date$()` command. If there is no date stored, the result is an empty string.

```
Func FileDate$({dayF%, {monF%, {yearF%, {order%, {sep$}}}}});
```

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.

`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()`

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%}}}});
```

`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 directory/folder set by the Edit menu preferences or by 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). The `upt%`, `tpa%` and `maxT` arguments must be supplied. This file type lets you create new Spike2 files from external data for use with memory channels.
- 12 An XY view with one (empty) data channel. Use `XYAddData()` to add more data and `XYSetChan()` to create new channels.

`mode%` This optional argument determines how the new window is opened. The value is the sum of these flags. If the argument is omitted, its value is 0. The flags are:

- 1 Make the new window(s) visible immediately. If this flag is not set the window is created, but is invisible.
- 2 For data files, if the sampling configuration holds information for creating additional windows, use it. If this flag is not set, data files extract enough information from the sampling configuration to set the sampling parameters for the data channels.
- 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 maximum time base that you can display for the file. You should set this to the expected file size.

`nChan%` Only for type% = 7. This sets the number of channels in the new file in the range 32 to 100. If you omit this optional argument, 32 channels are used. Spike2 4.02 onwards can read files with other than 32 channels.

Returns It returns the view handle (or the handle of the lowest numbered duplicate for a data file with duplicate windows) or a negative error.

See also: `FileOpen()`, `FileSave()`, `FileSaveAs()`, `FileClose()`, `FilePathSet()`, `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 (not supported for `type%`=6 on the Macintosh 68k version).

Returns If a file opens without any problem, the return value is the view handle for the file (if multiple views open, it is the handle for the first time view created). For configuration files (`type%` of 6), the return value is 0 if no error occurs. If the file could not be opened, or the user pressed Cancel in the file open dialog, the returned value is a negative error code.

If multiple windows are created for a data file, you can get a list of the associated view handles using `ViewList(list%[],64)`.

See also: `FileDelete()`, `FileNew()`, `FileSave()`, `FileSaveAs()`, `FileClose()`, `BRead()`, `BReadSize()`, `BSeek()`, `BWrite()`, `BWriteSize()`, `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, this gets the current directory, 1 returns the path for Spike2 data files created by `FileNew()` as set by Edit menu Preferences, 2 gets the path to the Spike2 application, 3 returns the path for automatic file saving as set by the Sampling Configuration Automation tab.
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()`.

```
Func FilePathSet(path${,opt%{, make%|text$}});
```

`path$`	A string holding the new path to the directory. The path must conform to the rules for path names on the host system and be less than 255 characters long. If the path is empty a dialog opens for the user to select an existing directory/folder.
`opt%`	If omitted or zero, this sets the current directory. 1 sets the path for Spike2 data files created by `FileNew()`, 2 is ignored, 3 sets the automatic file naming path.
`make%`	If this is present and non-zero, the command will create the directory/folder if all elements of the path exist except the last. You cannot use this option if `path$` is blank.
`text$`	Optional prompt for use with the dialog.
Returns	Zero if the path was set, or a negative error code.

See also:`FileList()`, `FileName$()`, `FilePath$()`

FilePrint()

This function is equivalent to the File menu Print command. It prints some, or all of the current view to the printer. 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.

```
Func FilePrint({from {, to}});
```

`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.
Returns	The function returns 0 if all went well; otherwise it returns a negative error.

The format of the printed output is based on the screen format of the current view. Beware that for time and result views the output could be many (very many) pages long.

See also:`FilePrintScreen()`, `FilePrintVisible()`

FilePrintScreen()

This function is equivalent to the File menu Print Screen command. It prints all visible 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({pTtl${, vTtl%{, box%{, scTxt%{, land%}}}}});
```

`pTtl$`	This sets the page title string to print at the top of a page. If omitted or an empty string, there is no page title.
`vTtl%`	Set 1 or higher to print a title above each view, omitted or 0 for no title.
`box%`	Set 1 or higher for a box around each view. If omitted, or 0, no box is drawn.
`scTxt%`	Set 1 or higher to scale text differently in the x and y directions to match the original. If omitted or 0 scale both directions by the same scale factor. **Warning:** this was new in version 3, version 2 scripts need changing.
`land%`	Set the page orientation: 1 or higher for landscape, omitted or 0 for portrait.
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()
```

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 Quit/Exit command. If there is any unsaved data you are asked if you wish to save it before the application closes. If the user cancels the operation (because there were files that needed saving), the script terminates, but the 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 or with time views unless they have been sampled and not 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:`FileOpen()`, `FileSaveAs()`, `FileClose()`, `SampleAutoFile()`, `SampleAutoName$()`

FileSaveAs()

This function saves the current view as a file on disk either in its native format, or as text 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 is also used to name data files after sampling, but read the remarks about time views below.

```
Func FileSaveAs(name${, type%{, yes%{, text${, nChan%}}}});
```

`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):

-1 Save in the native format for the view. Use this to save and name a Spike2 data file immediately after it has been sampled. 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

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.

0 Save sections set by `ExportChanList()` to a new Spike2 data file.

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 `ExportTextFormat()` and `ExportChanFormat()`. This also can be used 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. You cannot save a view as a bitmap file from the script.

6 Save sampling configuration in a configuration file.

12 Save as XY view (XY views only).

`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$` An optional prompt displayed as part of the file dialog to prompt the user. From version 3.14 onwards the File Save As dialog will appear if `text$` is not empty.

`nChan%` Used when `type%` is 0 to set the number of channels in the exported file in the range 32-100. If omitted, the file has the same number of channels as the source file. Only Spike2 4.02 and onwards can read files with more than 32 channels.

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 time view that has just been sampled as described above for `type%` set to -1. However, if automatic filing is set with `SampleAutoFile()` or by the sampling configuration, the file is saved automatically as soon as sampling finishes and 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: `EditCopy()`, `ExportChanFormat()`, `ExportChanList()`, `ExportTextFormat()`, `FileClose()`, `FileNew()`, `SampleAutoFile()`, `SampleAutoName$()`, `SampleStart()`, `SampleStatus()`, `SampleStop()`

FileTime$()

Since version 4.03, the time and date at which sampling started is saved in the data file. This function returns a string holding the time. Use `FileTimeDate()` to get the date as numbers. The current view must be a time view. The arguments are exactly the same as for the `Time$()` command. If there is no time stored, the result is an empty string.

```
Func FileTime$({tBase%, {show%, {amPm%, {sep$}}}});
```

See also:`Time$()`, `FileDate$()`, `FileTimeDate()`

FileTimeDate()

Since version 4.03, the time and date at which sampling started is saved in the data file. This function returns the time and date as numbers. Use `FileTime$()` and `FileDate$()` to get the result as strings. The current view must be a time view. The arguments are exactly the same as for the `TimeDate()` command. If there is no date stored, the returned values are all zero.

```
Proc FileTimeDate(&s%, {&m%,{&h%,{&d%,{&mon%,{&y%,{&wDay%}}}}}});
Proc FileTimeDate(now%[])
```

See also:`Date$()`, `MaxTime()`, `Seconds()`, `Time$()`, `TimeDate()`

FiltApply()

Applies a set of filter coefficients or a filter in the filter bank to a source waveform channel in the current time view and places the resulting waveform in a destination channel. If there is a large amount of data in the source channel, you should filter directly to a disk based channel, rather than a memory channel to avoid running out of memory.

Each output point is generated from the same number of input points as there are filter coefficients. Half these point are before the output point, and half are after. Where more data is needed than exists in the source file (for example at the start and end of a file and where there are gaps), extra points are made by duplicating the nearest valid point.

```
Func FiltApply(n%|coef[], dest%, srce%, sTime, eTime {,scale%})
```

`n%`	Index of the filter in the filter bank to apply in the range 0-11, or
`coef[]`	An array holding a set of FIR filter coefficients to apply to the waveform.
`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 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.
`scale%`	If this is present and non-zero, the scale and offset values of the output are optimised to give the best possible representation of the filtered data, but the filtering operation takes twice as long. If there is data in the channel already, this data may lose some of its accuracy through this process. If you choose not to re-scale, the channel's scale and offset will be unchanged and you run a slight risk of the waveform going outside the 16-bit range allowed for 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()`, `FiltAtten()`, `FiltCalc()`, `FiltComment$()`, `FiltCreate()`, `FiltInfo()`, `FiltName$()`, `FiltRange()`

FiltAtten()

This set the desired attenuation for a filter in the filter bank. When `FiltApply()` or `FiltCalc()` is used, the number of coefficients needed to achieve this attenuation will be generated. A value of zero sets the attenuation back to the default (-65 dB).

```
Func FiltAtten(index%{, dB})
```

`index%`	Index of the filter in the filter bank to use in the range 0-11.
`dB`	If present and negative, this is the desired attenuation for stop bands in the filter.
Returns	The desired attenuation for a filter at the time of the call.
See also:	`FiltApply()`, `FiltCalc()`, `FiltComment$()`, `FiltCreate()`, `FiltInfo()`, `FiltName$()`, `FiltRange()`

FiltCalc()

The calculation of filter coefficients can take an appreciable time. This routine forces the calculation of a filter for a particular sampling frequency if it has not already been done. If you do not force the calculation, you can still use `FiltApply()` to apply a filter. However, the coefficient calculation will then be done at the time of filter application, which may not be desirable if the filtering operation is time critical.

```
Func FiltCalc(index%, sInt{, coeff[]{, &dBGot {, nCoef%}}})
```

`index%`	Index of the filter in the filter bank to use in the range 0-11.
`sInt%`	The sample interval of the waveform you are about to filter. This is the value returned by `BinSize()` for a waveform channel.
`coeff`	An array to be filled with the coefficients used for filtering. If the array is too small, as many elements as will fit are set. The maximum size needed is 511 (it was 255 in all Spike2 versions before 4.01).
`dBGot`	If present, returns the attenuation attained by the filter coefficients.
`nCoef%`	If present, sets the number of coefficients used in the calculation (use an even number for a full differentiator and an odd number for all other filter types).
Returns	The number of coefficients generated by the filter.

An example

Suppose the first filter in the bank (index 0) is a low pass filter with the pass band edge at 50 Hz. If we know that we will need to filter a channel 4 (sampled at 200 Hz) with this filter, we may want to calculate the coefficients needed in advance:

```
FiltCalc(0, BinSize(4));
```

This will calculate a filter corresponding to the specification of filter 0 for a sampling frequency of 200 Hz with an attenuation in the stop band of at least the current desired attenuation value for this filter.

Constraints on filters

The calculation of coefficients is a complex process and can produce silly results due to floating point rounding errors in some situations. To ensure that you will always get a useful result there is a limit to how small and how big a transition gap can be relative to the sampling frequency. There is a similar limit on the width of a pass or stop band:

- The transition gap and the width of a pass or stop band cannot be smaller than 0.0025 of the sampling frequency.
- The transition gap cannot be larger than 0.12 of the sampling frequency.

This function always calculates a set of coefficients, but may alter the filter specification in order to do it (these changes are temporary, see later). This can happen in two cases:

1. If the sampling frequency is such that to produce the filter, the transition gap and/or pass and stop band widths are outside their limits, the widths are set to the limits before calculating the filter. In our 50 Hz low pass filter example, if we calculate it with respect to a 12 kHz sampling frequency, the minimum pass band width is 12000*0.0025 = 30 Hz. So, the filter would be changed to a 60 Hz low pass filter.
2. If half the sampling frequency (the Nyquist frequency) is less than an edge of a pass or stop band, certain attributes of the filter are lost. In our 50 Hz low pass filter example, if we tried to calculate with a sampling frequency of 80 Hz, we would see that the Nyquist frequency is 40 Hz. No frequency above 40 Hz can be represented in a waveform sampled at 80 Hz, so a 50 Hz low pass filter is equivalent to an “All pass” filter. The filter specification will be altered to reflect this before calculating.

Any changes made to a filter specification to accommodate a particular calculation are made with reference to the original specification, not the specification that was last used for a calculation.

See also:`FiltApply()`, `FiltAtten()`, `FiltComment$()`, `FiltCreate()`, `FiltInfo()`, `FiltName$()`, `FiltRange()`

FiltComment$()

This function gets and sets the comment associated with a filter in the filter bank.

```
Func FiltComment$(index% {, new$})
```

`index%`	Index of the filter in the filter bank to use in the range 0-11.
`new$`	If present, sets the new comment.
Returns	The previous comment for the filter at the index.

See also:`FiltApply()`, `FiltAtten()`, `FiltCalc()`, `FiltCreate()`, `FiltInfo()`, `FiltName$()`, `FiltRange()`

FiltCreate()

This function creates a filter in the filter bank to the supplied specification and gives it a standard name and comment.

```
Func FiltCreate(index%, type%{, trW{, edge1{, edge2{, ...}}}})
```

`index%`	Index of the new filter in the filter bank in the range 0-11. This action replaces any existing filter at this index.
`type%`	The type of the filter desired (see table).
`trW`	The transition width of the filter. This is the frequency interval between the edge of a stop band and the edge of the adjacent pass band.
`edgeN`	These are a list of edges of pass bands in Hz. (see table).
Returns	0 if there was no problem or a negative error code if the filter was not created.

This table shows the relationship between different filter types and the meaning of the corresponding arguments. The numbers in brackets indicate the nth pass band when there is more than 1. An empty space in the table means that the argument is not required.

`type%`	Name	`trW`	`edge1`	`edge2`	`edge3`	`edge4`
0	All stop					
1	All pass					
2	Low pass	Yes	High			
3	High pass	Yes	Low			
4	Band pass	Yes	Low	High		
5	Band stop	Yes	High(1)	Low(2)		
6	Low pass differentiator	Yes	High			
7	Differentiator					
8	1.5 Band Low pass	Yes	High(1)	Low(2)	High(2)	
9	1.5 Band High pass	Yes	Low(1)	High(1)	Low(2)	
10	2 Band pass	Yes	Low(1)	High(1)	Low(2)	High(2)
11	2 Band stop	Yes	High(1)	Low(2)	High(2)	Low(3)

The values entered correspond to the text fields shown in the Filter edit dialog box.

See also:`FiltApply()`, `FiltAtten()`, `FiltCalc()`, `FiltComment$()`, `FiltInfo()`, `FiltName$()`, `FiltRange()`

FiltInfo()

Retrieves information about a filter in the bank.

```
Func FiltInfo(index%{, what%})
```

`index%` Index of the filter in the filter bank to use in the range 0-11.

`what%` Which bit of information about the filter to return:

-2	Maximum `what%` number allowed
-1	Desired attenuation
0	type (if you supply no value, 0 is assumed)
1	Transition width
2-5	`edge1-edge4` given in `FiltCreate()`

Returns The information requested as float.

See also: `FiltApply()`, `FiltAtten()`, `FiltCalc()`, `FiltComment$()`, `FiltCreate()`, `FiltName$()`, `FiltRange()`

FiltName$()

This function gets and/or sets the name of a filter in the filter bank.

```
Func FiltName$(index% {, new$})
```

`index%` Index of the filter in the filter bank to use in the range 0-11.

`new$` If present, sets the new name.

Returns The previous name of the filter at that index.

See also: `FiltApply()`, `FiltAtten()`, `FiltCalc()`, `FiltComment$()`, `FiltInfo()`, `FiltCreate()`, `FiltRange()`

FiltRange()

Retrieves the minimum and maximum sampling rates that this filter can be applied to without the specification being altered. See the `FiltCalc()` command, *Constraints on filters* for more information.

```
Proc FiltRange(index%, &minFr, &maxFr)
```

`index%` Index of the filter in the filter bank to use in the range 0-11.

`minFr` Returns the minimum sampling frequency you can calculate the filter with respect to so that no transition width is greater than the maximum allowed and 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 place 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 (was 255 before version 4.01).
`nGrid`	The grid density for the calculation. If omitted or set to 0, the default density of 16 is used. This sets the density of test points in internal tables used to search for points of maximum deviation from the filter specification. The larger the value, the longer it takes to compute the filter. There is seldom any point changing this value unless you suspect that the program is missing the peak points.
`extFr`	An array to hold the list of extremal frequencies (the list of frequencies within the bands which have the largest deviation from the desired filter). If there are `n%` coefficients, there are `(n%+1)/2` extremal frequencies.

The parameters passed in must be correct or a fatal error results. Errors include: overlapping band edges, band edges outside the range 0 to 0.5, too many coefficients, differentiator slope less than 0, if not a differentiator the band function must lie between 0 and 1, the band weight must be greater than 0.

For example, to create a low pass filter with a pass band from 0 to 0.3 and a stop band from 0.35 to 0.5, and no return of the ripple, you would set up `param` as follows:

```
var param[4][2]      'No return of ripple, 2 bands
para[0][0] := 0;     'Starting frequency of pass band
para[1][0] := 0.3;   'Ending frequency of pass band
para[2][0] := 1;     'Desired gain (unity)
para[3][0] := 1;     'Give this band a weighting of 1

para[0][1] := 0.35;  'Starting frequency of stop band
para[1][1] := 0.5;   'Ending frequency of stop band
para[2][1] := 0;     'Desired gain of 0 (stop band)
para[3][1] := 10;    'Give this band a weighting of 10
```

See also:`ArrFilt()`, `FiltApply()`, `FiltCalc()`, `FIRQuick()`, `FIRResponse()`

FIRQuick()

This function creates a set of filter coefficients in the same way the `FIRMake()` does, but many of the parameters are optional, allowing the most common filters to be created with a minimal specification.

```
Func FIRQuick(coef[], type%, freq {, width {, atten}})
```

`coef`	An array into which the FIR filter coefficients are placed. The size of this array should be 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).

See also:`ArrFilt()`, `FiltApply()`, `FiltCalc()`, `FIRMake()`, `FIRResponse()`

FIRResponse()

This function retrieves the frequency response of a given filter as amplitude or in dB

```
Func FIRResponse(resp[], coef[]{, as%{, type%}});
```

`resp`	The array to hold the frequency response. This array will be filled regardless of its size. The first element is the amplitude response at 0 Hz and the last is the amplitude response at the Nyquist frequency. The remaining elements are set to the response at a frequency proportional to the element position in the array.
`coef`	The coefficient array calculated by `FIRMake()`, `FIRQuick()` or `FiltCalc()`
`as%`	If this is 0 or omitted, the response is in dB (0 dB is unchanged amplitude), otherwise as linear amplitude (1.0 is unchanged).
`type%`	If present, informs the command of the filter type. The types are the same as those supplied for `FIRQuick()`. 0=Low pass, 1=High pass, 2=Band pass, 3=Band stop and 4=Differentiator. If a type is given, the time to calculate the response is halved. If you are not sure what type of filter you have, or you have type not covered by the `FIRQuick()` types, then do not supply a type to this command.

See also:`ArrFilt()`, `FiltCalc()`, `FIRMake()`, `FIRQuick()`

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, **m** is the gradient of the line and **c** is the y axis intercept when x is 0.

```
Func FitLine(chan%, start, finish, &grad{, &inter{, &corr}});
```

`chan%`	A channel 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.

FontGet()

This function gets the name of the font, and its characteristics for the current view.

```
Func FontGet(&name$, &size, &style%);
```

`name$`	This string variable is returned holding the name of the font.
`size`	The real number variable is returned holding the point size of the font.
`style%`	Returned holding the style: 0=Normal, 1=Italic, 2=Bold, 3=Bold and Italic.
Returns	The function returns 0 if all was well or a negative error code. If an error occurs, the variables are not changed.

See also:`FontSet()`

FontSet()

This function sets the font for the current view. It does not cause an immediate redraw; use `Draw()` to force one. Text-based views (text, sequencer and script) should be set to non-proportionally spaced fonts only. If you set proportional fonts, the caret position in the view will be incorrect and the text will be difficult to edit.

```
Func FontSet(name$|code%, size, style%);
```

`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.
`style%`	The type style: 0=Normal, 1=Italic, 2=Bold, 3=Bold and Italic.
Returns	The function returns 0 if the font change succeeded, or a negative error code.

See also:`FontGet()`

Frac()

Returns the fractional part of a real number or converts a real array to its fractional parts.

`Func Frac(x|x[]);`

`x`	A real number or an array of reals.
Returns	For arrays, it returns 0 or a negative error code. If `x` is not an array it returns a real number equal to `x-Trunc(x)`. `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()

This command is used to set the view that is nearest to the top and makes it the current view. It is the view that would have the focus if all dialogs were removed. You can use this to find out the front view, or to set it. When a view becomes the front view, it is moved to the front unless it is already there. If an invisible or iconised view is made the front view, the view is made visible automatically, (equivalent to `WindowVisible(1)`).

`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:`View(), WindowVisible()`

Grid()

This function turns the background grid on and off for the current time, result or XY view. It also returns the state of the grid.

`Func Grid({on%});`

`on%`	Optional, sets the grid state. 0 = on, 1 = off, omit for no change.
Returns	The state of the grid at the time of the call, or a negative error code. Changes made by this function do not cause an immediate redraw.

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.

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.

`Func HCursor(num% {,where {,chan%}});`

`num%`	The cursor to use. It is an error to attempt this operation on an unknown cursor.
`where`	If this parameter is given it sets the new position of the cursor.

`chan%` If this parameter is given, it sets the channel number. In XY or result views you should omit this argument or use 1.

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 and result views return 1 as the channel number.

See also: `HCursor()`, `HCursorDelete()`, `HCursorLabel()`, `HCursorLabelPos()`, `HCursorNew()`, `HCursorRenumber()`

HCursorDelete()

This deletes the designated horizontal cursor. It is not an error to delete an unknown cursor; it just has no effect.

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

`num%` The number of the cursor to delete. If this is omitted, the highest numbered cursor is deleted. Set -1 to delete all horizontal cursors (version 3.08 onward).

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 command sets (or gets) the cursor label style for the current view.

```
Func HCursorLabel({style%{, num%{, form$}}})
```

`style%` The cursor style. Cursors can be annotated with a position or the cursor number or a user-defined style. The styles are: 0=Neither, 1=Position, 2=Number, 3=Both, 4=User-defined (not Macintosh yet). Unknown styles cause no change.

`num%` The cursor number for style 4. 0 means all cursors, 1-4 for a single cursor.

`form$` The label string for style 4. The string has replaceable parameters `%p`, and `%n` for position and number. We also allow `%w.dp` where `w` and `d` are numbers that set the field width and decimal places. You cannot read back a label format string.

Returns The previous cursor style. If you omit `style%`, the style does not change.

See also: `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 4 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 and result views.
`where`	An optional argument setting the cursor position. If this is omitted, the cursor is placed in the middle of the y axis or at zero if there is no y axis.
Returns	It returns the horizontal cursor number or 0 if all cursors are in use.

See also: `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()

The help available depends on the system. The Windows version uses the standard Windows help system. The Macintosh version may provide help through Apple Guide in future releases.

`Func Help(topic%|topic$ {,file$});`

`topic%`	A numeric code for the help topic. These codes are assigned by the help system author. Code 0 changes the default help file to `file$`.
`topic$`	A string holding a help topic keyword or phrase to look-up.
`file$`	If this is omitted, or the string is empty, the standard Spike2 help file is used. If this holds a filename, this filename is used as the help file.
Returns	1 if the help topic was found, 0 if it was not found, -1 if the help file was not found.

The Windows SDK has some help authoring tools, and third party tools are available.

Inkey()

This function 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}});
```

`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.

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$(),Str$(),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$` These label strings create buttons, from right to left, in the tool bar. If no labels are given, one label is displayed with the text `"OK"`. The maximum number of buttons is 17. Buttons can be linked to the keyboard using `&` and by adding a vertical bar followed by a key code to the end of the label. See the documentation for `label$` 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 is returned 0 if the transition is low to high, and 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$(), DelStr$(), InStr(), Left$(), Len(), Mid$(), Print$(), ReadStr(), Right$(), 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:`Asc(), Chr$(), DelStr$(), InStr(), LCase$(), Len(), Mid$(), Print$(), ReadStr(), Right$(), Str$(), UCase$(), Val()`

Len()

This function returns the length of a string or the size of a one dimensional array.

```
Func Len(text$);
Func Len(arr[]);
```

`text$` The text string.

`arr[]` A one dimensional array. It is an error to pass in a two dimensional array.

Returns The length of the string or the array as an integer.

You can find out the size of each dimension of a two dimensional array as follows:

```
proc something(arr[][])    'function passed a 2-d array
var n%, m%;
n% := Len(arr[][0]);       'get size of first dimension
m% := Len(arr[0][]);       'get size of second dimension
return;
end;
```

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

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

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. You need this if you are to size the log window, or make it the current or front window, or to use the Edit commands to clear it. The log window is created by the application and is the destination for `PrintLog()`. The Log window is a simple text window. You can hide it by clicking in the go away box. You can clear the contents with the Edit menu Clear command.

```
Func LogHandle()
```

Returns The view handle of the log window.

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 with. 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()`

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.

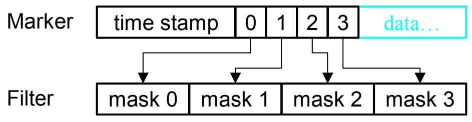
See also:`LastTime()`, `MarkEdit()`, `MarkMask()`, `NextTime()`

MarkMask()

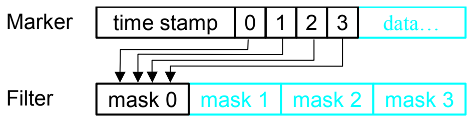
This function sets the mask for a marker, WaveMark, TextMark or RealMark channel. Each data item in one of these channels has four marker codes. Each marker code can have a value from 0 to 255.

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 is 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 two command variants, the first sets the filter codes, the second the filter mode:

```
Func MarkMask(chan%, layer%, set%, code%|code$ {,code%|code$...});
Func MarkMask(chan%, mode%);
```

`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.
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. Beware that mode 1 and the command to set modes are not present in versions of Spike2 before 3.13.

See also: `ChanDuplicate()`, `LastTime()`, `MarkEdit()`, `MarkInfo()`, `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()`

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()

This function adds or changes a measurement channel in an XY view created with `MeasureToXY()` using the settings previously defined by using `MeasureX()` and `MeasureY()`. The XY view must be the current view. This command implements some of the functionality of the XY plot setting dialog.

This command may be used in the future to add channels to tabulated measurements or to RealMark channels in a time view.

```
Func MeasureChan(chan%, name$ {,pts%|id% {,ch%}});
```

`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.
`id%`	Reserved for future implementations to identify the process command to which this command applied.
`ch%`	Reserved for future implementations to identify the output RealMark channel to hold the result.

Returns The channel number these `settings` were applied to or a negative error code.

See also:`CursorActive()`, `MeasureToXY()`, `MeasureX()`, `MeasureY()`

MeasureToXY()

This function creates a new XY view and an associated measurement process for channels in the current time view and sets the cursor 0 iteration method. It creates one output channel with a default measurement. You can edit the measurement method and add additional channels using `MeasureX()`, `MeasureY()` and `MeasureChan()`. This command implements some of the functionality of the XY plot setting dialog. Use `Process()` to generate the plot. The new view is the current view and is invisible. Use `WindowVisible(1)` to make it visible.

```
Func MeasureToXY(mode%,chan%,min|exp$ {,lv {,hw {,flgs%{,qu$}}}});
```

`mode%` This is the cursor 0 iteration mode. The modes are the same as for the `CursorActive()` command, but not all modes can be used. Valid modes are:

4 Peak find	8 Falling threshold	14 Data points
5 Trough find	12 Slope peak	16 Expression
7 Rising threshold	13 Slope trough	17 Turning point

`chan%` This is 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 sets the threshold level for threshold modes and the peak size for peak and trough modes. This argument is in the y axis units of channel `chan%` (or y axis units per second for modes 12 and 13). This argument is ignored and should be 0 or omitted for modes that do not require it.

`hw` This sets the hysteresis level in y axis units for threshold modes and the width in seconds for slope measurements. This argument is ignored and should be 0 or omitted for modes that do not require it.

`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.

Returns The function result is an XY view handle or a negative error code.

The following example generates a plot of the peak values from 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()

The `MeasureX()` and `MeasureY()` functions set the x and 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. This command implements some of the functionality of the XY plot setting dialog.

```
Func MeasureX(type%, chan%, expr1$ {,expr2$ {,width}});
Func MeasureY(type%, chan%, expr1$ {,expr2$ {,width}});
```

`type%` This sets the x or y measurement type. Values less than 100 match those for the `ChanMeasure()` command (types 5 and 1 are identical for this command).

1 Area	5 Area (scaled)	9 Minimum	13 Absolute maximum
2 Mean	6 Curve area	10 Peak to Peak	14 Peak
3 Slope	7 Modulus	11 RMS Amplitude	15 Trough
4 Sum	8 Maximum	12 Standard deviation	

Values from 100 up are:

100 Value	104 Fit coefficient (reserved)
101 Value difference	105 User entered value
102 Time	106 Expression
103 Time difference	

`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$` This expression evaluates as the start time for measurements over a time range, as the position for time and value measurements and as the expression used for measurement type 106.

`expr2$` This expression evaluates as the end time for measurements over a time range and as the reference time for position for single-point measurements and differences. Set this to 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()`, `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 100 memory channels belonging to a file.

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

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

1 Waveform	4 Level (Event+-)	7 RealMark
2 Event (Event-)	5 Marker (default)	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. This is ignored for all other channel types and should be set to 0.

`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.

Before Spike2 version 4.03, `binsz` was rounded to the nearest multiple of the underlying time resolution times *time per ADC* for the data file.

`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.

Returns Either the channel number of the newly created channel, or 0 if there are no free channels, or a negative error code.

Channels created in this way are given default titles, units and comments. You can set these with the `ChanTitle$()`, `ChanUnits$()`, `ChanComment$()`, `ChanScale()` and `ChanOffset()` routines. The following code creates a copy of channel `wChan%` (a waveform channel) as a memory channel:

```
func CopyWave%(wChan%) 'Copy waveform to a memory channel
var mc%;
if ChanKind(wChan%)<>1 then return 0 endif; 'Not a waveform!
mc% := MemChan(1,0,BinSize(wChan%));     'Create waveform channel
if mc%>0 then                            'Created OK?
   ChanScale(mc%, ChanScale(wChan%));    'Copy scale...
   ChanOffset(mc%, ChanOffset(wChan%));  '...and offset...
   ChanUnits$(mc%, ChanUnits$(wChan%));  '...and units
   ChanTitle$(mc%, "Copy");              'Set our own title
   ChanComment$(mc%, "Copied from channel "+Str$(wChan%));
   MemImport(mc%, wChan%, 0, MaxTime()); 'Copy data
   ChanShow(mc%);                        'display new channel
endif;
return mc%;   'Return the new channel number
end;
```

See also: `ChanNew()`, `MemDeleteItem()`, `MemDeleteTime()`, `MemGetItem()`, `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: `MemChan()`, `MemDeleteTime()`, `MemGetItem()`, `MemImport()`, `MemSave()`, `MemSetItem()`

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, that 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 points 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()`, `MemChan()`, `MemDeleteItem()`, `MemDeleteTime()`, `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 All others	Data copied, but must match sample interval Not available, see `EventToWaveform()`
Event	Waveform All others	Can extract event times Times are extracted from the channel.
Level	Waveform All others	Can extract event times Times are extracted from the channel. First time in destination is assumed to be a low to high transition.
Marker	Waveform Event, Level All others	Can extract event times, coded for peak/trough etc. Marker codes all set to 0. Marker codes are copied.
TextMark	Waveform Event, Level TextMark All others	Can extract event times, coded for peak/trough etc. Copies times, marker codes set 0, empty strings Copies all data, strings may be truncated if too long. Copies marker information, empty strings
RealMark	Waveform Event, Level RealMark WaveMark All others	Can extract event times, coded for peak/trough etc. Times copied, marker codes and reals set to 0 Copied, real data truncated or zero padded as needed Copied, waveform to reals, padded/truncated Marker portion copied, reals filled with 0
WaveMark	Waveform Event, Level WaveMark All others	Special option, event channel marks waveforms Copies times, marker codes set to 0, waveform set 0 Copies all data, waveform truncated/zero padded 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%` This was added at version 4.10. 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()`, `MemDeleteItem()`, `MemDeleteTime()`, `MemGetItem()`, `MemSave()`, `MemSetItem()`

MemSave()

This function writes the contents of 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. Memory channels are not saved when the associated data file closes.

```
Func MemSave(chan%, dest%{, type% {, query%}});
```

`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.

Returns The number of items written, or a negative error code.

See also: `ChanDelete()`, `ChanWriteWave()`, `MemChan()`, `MemDeleteItem()`, `MemDeleteTime()`, `MemGetItem()`, `MemImport()`, `MemSetItem()`

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%`	This is an integer array of at least 4 elements that hold the marker codes for the channel. If the channel does not require marker codes, this argument is ignored. If this parameter is omitted for a channel with markers, 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.

The output string will be presented as one line if it is short enough, otherwise it will be split into multiple lines. 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$()`, `UCase$()`, `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%}}}}}});
```

`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 sets the drawing mode in which to find the minimum and maximum. If `mode%` is absent or inappropriate, the display mode is used. This parameter is ignored in a result view. The modes in a time view are:

0	The standard mode for the channel.
1	Dots mode for events. The `dotSz%` argument can be used.
2	Lines mode.
3	Waveform mode. This is the only mode for waveform channels.
4	WaveMark mode.
5	Rate mode. The `binSz` argument sets the width of each bin.
6	Mean frequency mode. `binSz` sets the time period.
7	Instantaneous frequency mode. `dotSz%` can be used.
8	Raster mode. `trig%` sets the trigger channel, `dotSz%` is used.

`binSz` This sets the width of the rate histogram bins and the smoothing period for mean frequency mode when specifying your own mode.

`trig%` The trigger channel for raster displays, level data raster displays are impossible.

`edge%` For level data event channels. It sets which edges of the level signal are used for mean frequency, instantaneous frequency and rate modes. The values are:

0	Use both edges (same as omitting the parameter).
1	Use rising edges.
2	Use falling edges

Returns Zero if all was well or a negative error code.

See also:`Min(), Max(), XYRange()`

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 is returned 0 if the transition is low to high, and 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. If you specify all channels, only visible channels are optimised. However, if you give the number of a hidden channel, it is optimised.

```
Proc Optimise(chan%{, start{, finish}});
```

`chan%`	The channel to optimise. We also allow -1 for all, -2 for visible and -3 for all selected 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()`

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()`

PlayWaveAdd()

This command adds a new area to the on-line play wave list. When you create a data file, Spike2 reserves 1401 memory for the area and transfers any stored waveform to it. It is set to play once and is not linked to any other area. The replay speed factor is set to 1.0 and the wave is set to non-triggered. Use this command before `FileNew()`.

There are three command variants. The first adds a wave from the current time view or from a Spike2 data file, the second adds a wave from a data array, and the third reserves 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$` variable to be used as a source of output data.
`wch%`	Either a single channel, or an array of channels to use as a data source for playing. There must be one source channel for each output channel set by the `dac%` variable. The channels can be either waveform or WaveMark data. The sample rate is taken from the sampling rate of the first channel in the list. If subsequent channels in the list have different rates, data is interpolated.
`mem%`	If present, and non-zero, the data is converted to a memory image and becomes independent of the data file. Otherwise, Spike2 stores the file name and extracts the data as required for sampling.
`path$`	This optional argument sets the name of the file to extract data from. If absent, the current view (which must be a time view) is used as the data source. The file must exist 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 2,000,000 bytes.

If you wish to link areas together or set the number of times the area is to be repeated or change the area speed factor, use one of the other `PlayWave...()` commands. The `SampleClear()` command 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: `FileNew()`, `PlayWaveChans()`, `PlayWaveCycles()`, `PlayWaveDelete()`, `PlayWaveEnable()`, `PlayWaveInfo$()`, `PlayWaveLabel$()`, `PlayWaveLink$()`, `PlayWaveRate()`, `PlayWaveSpeed()`, `PlayWaveTrigger()`, `SampleClear()`

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$`	The first character of the string identifies the area to be updated.
`data`	This is either an integer or a real array. If there is more than one DAC channel to play, the array must have the same number of rows as there are DACs, that is if there are three DACs, the array must be equivalent to `var data[n][3];` where `n` is the number of data points. If this is an integer array, the bottom 16 bits of each element is played through the DACs. If the data is in a real array, we assume that the full range of the DACs is ±5 Volts and that the data is the required output value in Volts. The data in the array is copied to 1401 memory. It is an error for the array size to be larger than the memory area in the 1401.
`offs%`	The offset within the data area, in data points per channel, to start writing. If the size of the data is such that the copy operation would extend beyond the end of the target area, the extra data is copied to the start of the area.
Returns	The function returns 0 if all went well or if you call when there is no sampling or sampling is stopping. It returns a negative error code if there is a problem or you have requested sampling to start and it hasn't yet started.
See also:	`PlayWaveCycles()`, `PlayWaveLink$()`, `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$`	The first character of the string identifies the play wave area.
`new%`	If present, this sets the number of cycles to play. The value 0 sets a very large number of cycles.
Returns	The number of cycles set at the time of the call.
See also:	`PlayWaveAdd()`, `PlayWaveEnable()`, `PlayWaveInfo$()`, `PlayWaveLabel$()`, `PlayWaveLink$()`, `PlayWaveRate()`, `PlayWaveSpeed()`, `PlayWaveStatus$()`, `PlayWaveStop()`

PlayWaveDelete()

This function deletes one or more play wave areas from the sampling configuration. If you do this while sampling is in progress it will not change the waves loaded to the 1401.

```
func PlayWaveDelete({keys$});
```

`keys$` If this is omitted, all play wave areas are deleted. If it is present, all areas with a key that is in this string are deleted. Case is significant for play areas; `"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 number of data points per channel that will be played out for this channel.

`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: `PlayWaveAdd()`, `PlayWaveChans()`, `PlayWaveCycles()`, `PlayWaveEnable()`, `PlayWaveLabel$()`, `PlayWaveLink$()`, `PlayWaveRate()`, `PlayWaveSpeed()`, `PlayWaveStatus$()`

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$()

You can link 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()`

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.

```
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. There is no guarantee that Spike2 will be able to play at the rate you request; as near a rate as possible to the requested rate is used. On-line changes are allowed.

```
func PlayWaveSpeed(key${, new{, wait%}});
```

`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.

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 to play. Positions are in terms of the number of points per channel in the area.
`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 are 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()`, `Frac()`, `Ln()`, `Log()`, `Max()`, `Min()`, `Rand()`, `Round()`, `Sin()`, `Sqrt()`, `Tan()`, `Trunc()`

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 this is present, the output field will be at least this wide. If this is present, it is a number that sets the minimum width of the output field. If the output is narrower than this, the field is padded on the left (on the right if the `-` flag was used) to this width with spaces (zeros if the `0` flag was used). The maximum width for numbers is 100.

`precision` This number sets the maximum number of characters to be printed for a string, the number of digits after the decimal point for `e` and `f` formats, the number of significant figures for `g` format and the minimum number of digits for `d` format (leading zeros are added if required). It is ignored for `c` format. There is no limit to the size of a string; numeric fields have a maximum `precision` value of 100.

`format` The format character determines how the argument is converted into text. Both upper 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.dddddde±xx{x}` where the number of `d`'s is set by the precision (which defaults to 6). A precision of 0 suppresses the decimal point unless the `#` flag is used. The exponent has at least 2 digits (in some implementations of 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. 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 commands prints to the log window. The syntax is identical to the `Print()` command, except that the output always goes to the end of the log window.

```
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 result or XY view. 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).MaxTime()` and `View(-1).Cursor(1)` to refer to time view times.

```
Func Process(sTime, eTime {,clear% {,opt%}});
```

`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.
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:`SetAverage(), SetEvtCrl(), SetINTH(), SetPhase(), SetPSTH(), SetPower(), SetResult(), SetWaveCrl(), ProcessAll(), Sweeps()`

ProcessAll()

This function is used in a time view to process all linked result views. It is as if a `Process()` command were used for each result view with the `clear%` and `opt%` parameters set to 0 and 1 unless a `Process()` command has been used on the result view, in which case the last used values of `clear%` and `opt%` apply.

```
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:`Process()`

ProcessAuto()

This is equivalent to the Automatic radio button in the Process Dialog for a New file. It sets up processing so that when the result view for which the function is used is given a chance to update, the parameters set by this command are used.

```
Func ProcessAuto(delay, mode% {,opt% {,last {,leeway}}});
```

`delay`	The minimum time between updates, in seconds.
`mode%`	0 = accumulate all data. 1 = clear the result, and process the data from 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.
`leeway`	For XY views only, it sets how close cursor 0 can be to the end of file.
Returns	0 or a negative error code.

See also:`ProcessTriggered(), Process()`

ProcessTriggered()

This is equivalent to the Process Dialog used with a New file in Triggered mode. It sets up processing so that when the result view for which the function is used is given a chance to update, the parameters set by this command are used.

```
Func ProcessTriggered(length, pre, chan%{,clear%{,opt%,{code%}}});
```

`length`	The length of data to process around each trigger point, in seconds.
`pre`	The pre-trigger time, in seconds.
`chan%`	The channel holding trigger events or markers.
`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.
`code%`	If present, and the channel is a marker or derived type, this holds the marker code to use as the trigger. Set this to -1 or omit it to use the current marker filter for the trigger channel.
Returns	0 or a negative error code.

See also:`ProcessAuto()`, `Process()`

Profile()

Spike2 stores information within the `HKEY_CURRENT_USER\Software\CED\Spike2` section of the system registry. The registry is organised as a tree of keys with lists of values attached to each key. If you think of the registry as a filing system, the keys are folders and the values are files. Keys and values are identified by case-insensitive text strings. This command can create and delete keys and store and read integer and string values, but only within the Spike2 section of the registry. (Added at version 4.08).

You can view and edit the registry with the `regedt32` program, which is part of your system. Select Run from the start menu and type `regedt32` then click OK. Please read the `regedt32` help information before making any registry changes. It is a very powerful program; careless use can severely damage your system.

Do not write vast quantities of data into the registry; it is a system resource and should be treated with respect. If you must save a lot of data, write it to a text or binary file and save the file name in the registry. If you think that you may have messed up the 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$`	This string sets the key to work on inside the `Spike2` section of the registry. If you use an empty string, the `Spike2` key is used. You can use nested keys separated by a backslash, for example `"My bit\\stuff"` to use the key `stuff` inside the key `My bit`. The key name may not start with a backslash. Remember that you must use two backslashes inside quote marks; a single backslash is an escape character. It is never an error to refer to a key that does not exist; the system creates missing keys for you.
`name$`	This string identifies the data in the key to read or write. If you set an empty name, this refers to the `(default)` data item for the key set by `key$`.
`val`	This can be either a string or an integer value. If `read` is omitted, this is the value to write to the registry. If read is present, this is the default value to return if the registry item does not exist.
`read`	If present, it must have the same type as `val`. This is a variable that is set to the value held in the registry. If the value is not found in the registry, the variable is set to the value of the `val` argument.

`Profile()` can be used with 1 to 4 arguments. It has a different function in each case:

1 The key identified by `key$` is deleted. All sub-keys and data values attached to the key and sub-keys are also deleted. Nothing is done if `key$` is empty.
2 The value identified by `name$` in the key `key$` is deleted.
3 The value identified by `name$` in the key `key$` is set to `val%` or `val$`.
4 The value identified by `name$` in the key `key$` is returned in `val%` or `val$`.

The following script example collects values at the start, then saves them at the end:

```
var path$, count%;
Profile("My data", "path", "c:\\work", path$); 'get initial path
Profile("My data", "count", 0, count%); 'and initial count
...                                     'your script...
Profile("My data","path", path$);       'save final value
Profile("My data","count", count%);     'save final count
```

Registry use by Spike2 The `HKEY_CURRENT_USER\Software\CED\Spike2` key contains the following keys that are used by Spike2:

BarList This key holds the list of scripts to load into the script bar when Spike starts.

Editor This key holds the editor settings for scripts, output sequences and general text editing.

PageSetup This key holds the margins in units of 0.01 mm for printing data views, and the margins in mm 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.
Event ports at rear	0=Events 0&1 on front panel, 1=on rear panel.
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.
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.

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 working set sizes in use are displayed in the Help menu About Spike2 dialog. Click the Help button 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 is a very powerful function. 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 DOS command prompt. Use `ProgRun()` to test the program status, `ProgKill()` to terminate it.

```
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()`

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 function returns pseudo-random numbers, and sets the seed for the random number generator. Spike2 seeds the generator when it starts with a number derived from the time, so if you require a repeatable sequence, you must set the seed. The sequence will not be the same on different implementations of Spike2.

```
Func Rand({seed});
```

`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.
Returns	A random number in the range 0 up to but not including 1. The numbers are evenly distributed. If a seed is given, a random number is still returned.

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

RasterAux()

Use `RasterSort()` or `RasterSymbol()` for version 5 compatibility. Each raster sweep has 6 auxiliary values, numbered 0 to 5. This function returns and optionally sets these values for 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.

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:`RasterAux()`, `RasterSet()`, `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:`RasterAux()`, `RasterGet()`, `SetEvtCrl()`, `SetPhase()`, `SetPSTH()`

RasterSort()

Each raster sweep has 2 values that can be used to sort the rasters. The `DrawMode()` command selects between time order and one of these values. This function returns and optionally sets these values for the current result view. Spike2 version 5 allows 4 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 access in the range 1 to 2. Values above 2 are treated as 2.
`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 4 symbols that can be displayed. This function returns and optionally sets these values for the current result view. Symbol values are times, in seconds relative to the start of the file. The 4 markers are: circle, cross, square, triangle. 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. Spike2 version 5 allows up to 8 symbols.

```
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 4. Values above 4 are treated as 4.
`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 the function extracts data from the string. In a successful call, each variable will be matched with a field in the string, and the value of the variable is changed to the value found in the field. A call to `Read()` with no arguments skips a line.
Returns	The function returns the number of fields in the text string that were successfully extracted and returned in variables, or a negative error code. Attempts to read past the end of the file produce the end of file error code. It is not considered an error to run out of data before all the variables have been updated. If this is a possibility you must check that the number of items returned matches the number you expected. If an array is passed in, it is treated as though it was the number of individual values held in the array.

The source string is expected to hold data values as real numbers, integer numbers and strings. Strings can be delimited by quote marks, for example `"This is a string"`, or they can be just text. However, if a string is not delimited by quotes, it is deemed to run to the end of the source string, so no other items can follow it.

The fields in the source string are separated by white space (tabs and spaces) and commas. Space characters are “soft” separators. You can have any number of spaces between fields. Tabs and commas are treated as “hard” separators. Two consecutive tabs or commas, or a tab and a comma (with or without intervening spaces), imply a blank field. When reading a field, the following rules are followed:

1. Space characters are skipped over
2. Characters that are legal for the variable into which data is to be read are extracted until a non-legal character or a separator or end of data is found. The characters read are converted into the variable type. If an error occurs in the translation, the function returns the error. Blank fields assigned to numbers are treated as 0. Blank fields assigned to strings produce empty strings.
3. Characters are then skipped until a separator character is found or end of data. If the separator is a space, it and any further spaces are skipped. If the next character is a hard separator it is also skipped.
4. If there are no more variables or no more data, the process stops, 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;
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
```
This is equivalent to the result

See also:`EditCopy()`, `FileOpen()`, `ReadStr()`, `Selection$()`

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()`, `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%`	If >0, the current time view starts to rerun unless it is already rerunning. If zero, rerun is cancelled for the time view. Omit to leave the state unchanged. Use –1 to return the current `sTime`, -2 to return `eTime` and –3 to return `scale`. The `sTime`, `eTime` and `scale` arguments are used when run% > 0.
`sTime`	Sets the start time for the rerun. If omitted, 0 is used.
`eTime`	Sets the end time of the rerun. If omitted `MaxTime()` is used.
`scale`	Sets the time scale for the rerun. If omitted, 1.0 is used. A value of 2 reruns twice as fast. Values from 0.01 to 100.0 are allowed.
Returns	If `run%` is positive or omitted the command returns the state at the time of the call: 0=not rerunning, 1=rerunning, -1= rerunning is not allowed. Negative values of `run%` return the current rerun arguments.

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$()`, `UCase$()`, `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 and closes the associated data, result and cursor views. If a result view derived from the data has been saved, the saved file remains. It is equivalent to the Abort button on the sampling control window.

`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()`

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 20 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 (version 3.11 and later). 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. If you call the command with no arguments it returns the number of buttons in the toolbar.

```
Func SampleBar({n% {, &get$}});
Func SampleBar(set$);
```

`n%` If set to -1, `get$` must be omitted and all buttons are cleared and the function returns 0. When set to the number of a button (the first button is 0), `get$` 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()`

SampleCalibrate()

This sets the waveform or WaveMark channel calibration. It changes the units, scale and offset fields only. See the Sampling Configuration dialog for a description of these fields.

```
Func SampleCalibrate(chan%, units$, scale, offset);
```

`chan%` The channel number of a waveform or WaveMark channel (1-100).

`units$` The units to use. If the string is longer than 5 characters only the first 5 are used.

`scale` The scale factor for the channel.

`offset` The offset for the channel.

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

See also: `SampleTimePerAdc()`, `SampleTitle$()`, `SampleUsPerTime()`, `SampleWaveform()`, `SampleWaveMark()`

SampleChannels()

This function returns and optionally sets the maximum number of channels in the sampling configuration. The next time you sample, the data file will have space for this number of channels. This command was added at version 4.03. Spike2 versions before 4.02 cannot read data files with other than 32 channels.

```
Func SampleChannels({nChan%});
```

`nChan%` If present, this sets the number of channels in the next data file created for sampling. The minimum number of channels is 32; if you set less than 32, 32 channels are selected. The maximum number of channels is currently 100. It is not an error to request more, but you are limited to the maximum.

Returns The number of channels set at the time of the call.

See also: `FileNew()`, `FileSaveAs()`, `SampleClear()`,

SampleClear()

This sets the sampling configuration to a standard state. The data file is set 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. All sample rate optimising is disabled (equivalent to `SampleOptimise(0,0,0)`).

```
Proc SampleClear();
```

See also: `SampleLimitSize()`, `SampleLimitTime()`, `SampleOptimise()`, `SampleTimePerAdc()`, `SampleUsPerTime()`

SampleComment$()

This function gets and sets the comment attached to a channel in the Sampling Configuration dialog.

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

`chan%` The channel number in the window (1 to 100).

`new$` If present, the new comment. If the comment is too long, it is truncated.

Returns The original comment, or an empty string if the channel number is illegal.

See also: `SampleDigMark()`, `SampleEvent()`, `SampleTextMark()`, `SampleTitle$()`, `SampleWaveform()`, `SampleWaveMark()`

SampleDigMark()

This adds the digital marker channel to the sampling configuration.

```
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.

```
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 sampling. This can be used to position, show and hide the output sequencer and sampling control panels. `App()` also returns control panel handles.

```
Func SampleHandle(which%);
```

`which%` Selects which view handle to return:
0 Main time window 1 Sampling control panel 2 Sequencer control panel

Returns The view handle or 0 if the view does not exist.

See also:`App()`, `View()`, `ViewList()`, `Window()`, `WindowVisible()`

SampleKey()

This procedure adds events to the keyboard marker channel. If there is no sampling, the procedure does nothing. If the output sequencer is running, and you add a key that corresponds to a key linked to a sequencer step, the sequencer jumps to the step.

```
Proc SampleKey(key$);
```

`key$` The first character of the string is added to the keyboard marker channel.

See also:`SampleAbort()`, `SampleReset()`, `SampleStart()`, `SampleStop()`, `SampleStatus()`, `SampleText()`, `SampleWrite()`

SampleLimitSize()

This corresponds to *Stop sampling when file size reaches* 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()`

SampleLimitTime()

This corresponds to *Stop sampling when time reaches* 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 function returns the time limit as it was at the time of the call. If the limit is disabled, the time is returned negated.

See also: `SampleClear()`, `SampleLimitSize()`, `SampleMode()`, `SampleTimePerAdc()`, `SampleUsPerTime()`

SampleMode()

This function sets and gets the sampling mode in the Sampling configuration dialog.

```
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 or Triggered mode.
-2	Returns the value of the `Every` field, valid for Timed sampling mode.
-3	Returns the T`rigger` 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.

See also: `SampleClear()`, `SampleLimitSize()`, `SampleLimitTime()`

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 or Slow channels.

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 (Quick or Slow) 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 can 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 this command after all the other sample setup commands so that you pay the time penalty for optimising once.

```
Func SampleOptimise(opt%, group%, type% {,usLo% {,usHi%}});
Func SampleOptimide(get%);
```

`opt%` This sets the optimise method. Values are:

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.

`group%` This controls the use of Quick channels. Values are:

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:
- 0 Works with all supported 1401s except a 1401*plus* with an old ADC.
- 1 Set for a 1401*plus* with an old ADC.
- 2 Set for a Power1401.
- 3 Set for a 1401plus or a micro1401.
- 4 Set for a Micro1401 mk II.

`usLo%` If present, it sets the low limit, in microseconds, for optimising microseconds per time unit (when `opt%` is 2). Values outside the range 2 to 1000 are ignored.

`usHi%` If present, it sets the high limit, in microseconds, for optimising microseconds per time unit. Values outside the range 2 to 1000 are ignored.

`get%` Use the function with one argument to read back the current settings. The values 0 to 4 read back the current values of `opt%`, `group%`, `type%`, `usLo%` and `usHi%`.

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()`

SampleReset()

This is used to abandon sampling, deleting any data that has been written to disk, and return to the state as if `FileNew()` had just been used to create a new data file.

```
Func SampleReset();
```

Returns 0 if the reset operation completed without a problem, or a negative error code.

See also: `SampleAbort()`, `SampleKey()`, `SampleStart()`, `SampleStop()`, `SampleStatus()`, `SampleText()`, `SampleWrite()`

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()`

SampleSequencer()

You can use this function to set the sequencer file to attach to the Sampling configuration. Use `SampleSequencer$()` to get the name of the current sequencer file.

```
Func SampleSequencer(new$);
```

`new$` The name of the sequence file. Pass an empty string to set no sequencer file.

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

See also: `SampleKey()`, `SampleSequencer$()`, `SampleSeqVar()`

SampleSequencer$()

This function returns the name of the sequencer file that is currently attached to the sampling configuration. Use `SampleSequencer()` to set the file.

```
Func SampleSequencer$();
```

Returns It returns the current sequencer file name, or an empty string if there is no file. The returned name includes the full path.

See also: `SampleKey()`, `SampleSequencer()`, `SampleSeqVar()`

SampleSeqVar()

This is used during sampling with an output sequence to get or set the value of an output sequencer variable. Values set before the sampling window exists are ignored. From 4.04, values set before `SampleStart()` set the initial variable values. Previously they took effect after sampling started.

```
Func SampleSeqVar(sVar%{, value%});
```

`sVar%`	The sequencer variable to set or read in the range 1 to 64.
`value%`	The value for the output sequencer variable. If present, the value of the variable is updated. If absent, the value of the variable is returned. 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%`	If this is 0 or omitted, sampling starts immediately, otherwise sampling waits for a trigger signal on the 1401 E3 input (Trigger input of the micro1401).
Returns	0 if all went well or a negative error code.

See also: `SampleAbort()`, `SampleKey()`, `SampleReset()`, `SampleStop()`, `SampleStatus()`, `SampleText()`, `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.

```
Func SampleStop();
```

Returns 0 if sampling stopped correctly or a negative error code.

See also: `SampleAbort()`, `SampleKey()`, `SampleReset()`, `SampleStart()`, `SampleStatus()`, `SampleText()`, `SampleWrite()`

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 as a text marker channel. 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.

```
Func SampleTextMark(max%{, port%{, term$
                    {, baud%{, bits%{, par%{, stop%{, hsk%}}}}}}});
```

`max%`	This sets 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 9, to use to read on-line TextMark data. If omitted, no serial data is read. When reading, characters with codes less than 32 are ignored unless they are the terminator character.
`term$`	Optional terminating character for serial line read. If omitted, `"\r"` is used (carriage return, character code 13).
`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 Baud rates from 50 to 115200 are supported. If you do not supply 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%`	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.
`hsk%`	This sets the handshake mode, sometimes called flow control. 0= no handshake, 1=hardware handshake, 2=XON/XOFF protocol. If omitted, 0 is set.
Returns	0 or a negative error code.

See also:`SampleText()`, `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 be at least 3 for Power1401 and 6 for 1401*plus* and micro1401. Lower values will cause sampling to fail. If the optimise method is set to anything other than 0 by `SampleOptimise()`, this call will have no effect as time per ADC will be recalculated.

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

`new%` The clock ticks per conversion in the range 1 to 32767. If this argument 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: `SampleClear()`, `SampleMode()`, `SampleOptimise()`, `SampleUsPerTime()`, `SampleWaveform()`, `SampleWaveMark()`

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()`

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, 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 standard Spike2 defaults (input in Volts). The title and comment of the channel are not changed.

```
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 0-31.

`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 adds a WaveMark channel to the sample list. The units, scale and offset fields are set to the Spike2 defaults (input in Volts). The channel title and comment are not changed. See the Sampling Configuration dialog description for sample rate details.

```
Func SampleWaveMark(chan%, port%, rate, size%, pre% {,ideal});
```

`chan%` The channel number to use for the new channel in the range 1 to 100.

`port%` The 1401 waveform port in the range 0-31.

`rate`	The estimated maximum sustained spike rate, used to allocate buffer space.
`size%`	The number of waveform samples to save for each WaveMark This must be an even number in the range 10 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. It is a fatal error for this to be outside the range 1.0 to 500000.0 Hz. This argument is new in version 4.
Returns	0 if all went well, or a negative error code.

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 (version 3.11 and later). 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 that control the new script. If omitted, 0 is used. The only flag defined now is 1 = run new script even if the current script ends in an error.

If the file you name does not exist when Spike2 tries to run it, nothing happens. If the nominated script is not already loaded, Spike2 will load it, run it and unload it.

See also:`App()`, `Profile()`

Seconds()

This returns the time in seconds. This is used for relative time measurements. You can also set the timer. If you want the time into sampling, use the `Maxtime()` function.

```
Func Seconds({set});
```

`set` If present, this sets the time in seconds. The time is 0 when Spike2 starts.

Returns The time in seconds. This is the value before any new time is set.

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 there is no text 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. Ports are automatically closed when a script ends, however it is good practice to close a port when your script has finished with it.

```
Func SerialClose(port%);
```

`port%` The serial port to close as defined for `SerialOpen()`.

Returns 0 or a negative error code

See also:`SerialOpen()`, `SerialWrite()`, `SerialRead()`, `SerialCount()`

SerialCount()

This counts the characters or items buffered in a serial port opened by `SerialOpen()`. Use this to detect input so your script can do other tasks while waiting for serial data. There is an internal buffer (belonging to Spike2) of 1024 characters per port that is filled when you use `SerialCount`. The size of this buffer limits the number of characters that this function can tell you about. To avoid character loss when you are not using a serial line handshake, do not buffer up more than a few hundred characters with `SerialCount`.

```
Func SerialCount(port% {,term$});
```

`port%` The serial port to use as defined for `SerialOpen()`.

`term$` An optional string holding the character(s) that terminate an input item.

Returns If `term$` is absent or empty, this returns the number of characters that could be read. If `term$` is set, this returns the number of complete items that end with `term$` that could be read.

See also:`SerialOpen()`, `SerialWrite()`, `SerialRead()`, `SerialClose()`

SerialOpen()

This function opens a serial port and configures it for use by the other serial line functions. It is not an error to call `SerialOpen` more than once on the same port. The serial routines use the host operating system serial line support. Consult your system documentation for information on serial line connections and baud rates limit.

```
Func SerialOpen(port%{, baud%{, bits%{, par%{, stop%{, hsk%}}}}});
```

`port%`	The serial port to use in the range 1 to 9. The number of ports depends on the computer. Two ports (1 and 2) are common on both PC and Macintosh systems.
`baud%`	This sets the serial line Baud rate (number of bits per second). The maximum character transfer rate is of order one-tenth this figure. All standard rates from 50 to 115200 Baud are supported. If you 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 this 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 when the function returns the number of characters returned.
`max%`	If present, it sets the maximum number of characters to read into each string or into the integer array. If a terminator is set, but not found after this many

characters, the function breaks the input at this point as if a terminator had been found. There is a maximum limit set by the 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 hanging up Spike2.

See also:`SerialOpen()`, `SerialWrite()`, `SerialCount()`, `SerialClose()`

SerialWrite()

This writes one or more strings or binary data to a serial port opened by `SerialOpen()`.

```
Func SerialWrite(port%, out$|out$[]|out%|out%[]{, term$});
```

`port%`	The serial port to write to as defined for `SerialOpen()`.
`out$`	A single string to write to the output or an array of strings to write. The return value is the number of strings written. If a time-out occurs, the function returns the number of complete strings sent before the time-out.
`out%`	A single integer or an integer array to write as binary. One value is written per integer. The output written depends on the number of data bits set for the port; 7-bit data writes as `out% band 127`, 8-bit data writes as `out% band 255`. The return value is 1 if the transfer succeeded.
`term$`	If present, it is written to the output port after the contents of `out%`, `out%[]` or `out$` or after each string in `out$[]`.
Returns	As defined above or a negative error code. If the output system becomes full, the function waits for one second before timing out.

See also:`SerialOpen()`, `SerialRead()`, `SerialCount()`, `SerialClose()`

SetXXXX commands

These commands create result views attached to the current time view or to the time window associated with the current result view. Apart from `SetResult()`, which creates a result view that is not dependent on a time view, these routines correspond with 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 duplicated, the first occurrence is used. It is an error if the resulting list is empty. Channels are created in the result view in the order they appear in the specification. The first channel in the list generates result view 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%}}});
```

`cSpc`	A channel list specifier defining channels to average. The channels must be waveform or WaveMark 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.
`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. Add 1 to display the mean data and not the sum. Add 4 to enable error bars. If `flags%` is omitted, the value 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 enable an auxiliary channel to take a measurement from each sweep and use this value to sort rasters and/or discard sweeps. See the Event correlation in the Analysis menu for details. See `RasterAux()` for a description of auxiliary values.

```
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 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.
`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 flag values: 1 to scale the result as spikes per second, 2 to enable raster displays, 8 for backwards `aCh%` event search. If omitted, 0 is used.
`aCh%`	Auxiliary measurement channel. The measured value sets auxiliary value 0 for sweep sorting. 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 auxiliary value 2, the symbol 1 position. If you omit `aCh%`, there is no auxiliary channel.
`Mn,Mx`	If these two values are present, they set the allowed range of measured values from `aCh%`. If the value for a sweep is outside this range, the sweep is discarded.
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()`

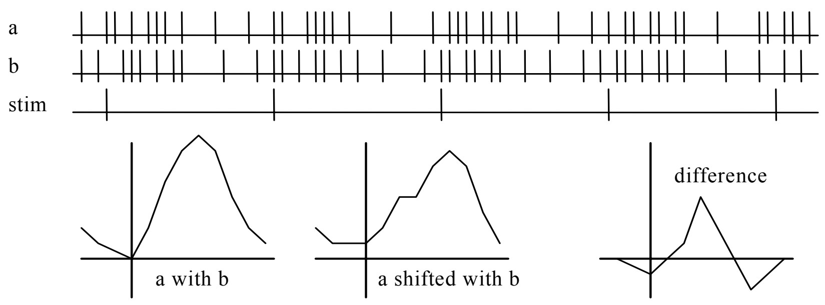
SetEvtCrlShift()

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 the result of an interval histogram analysis. 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. The function is:

```
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 bin for analysis, in seconds. Intervals shorter than this are counted for `Sweeps()`, but not included in the histogram. If `minInt` is 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 phase histogram with an optional raster display. The `Process()` command does the analysis. `Sweeps()` reports the number of cycles analysed. You can enable an auxiliary channel to take a measurement for each sweep and use this value to sort rasters and/or discard sweeps. See the description of the Analysis menu Phase histogram for details. See the `RasterAux()` command for the use of auxiliary values.

```
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 flag values: 2 to enable raster displays, 8 for backwards `aCh%` event search. If `flags%` is omitted, the value 0 is used.
`aCh%`	Auxiliary measurement channel. The measured value sets auxiliary value 0 for sweep sorting. 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 auxiliary value 2, the symbol 1 position. If you omit `aCh%`, there is no auxiliary channel.
`Mn,Mx`	If these two values are present, they set the allowed range of measured values from `aCh%`. If the value for a sweep is outside this range, the sweep is discarded.
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 ignored; there must be at least one valid channel. Channels with sampling rates 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 4096. 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 enable an auxiliary channel to measure a value for each sweep and use this to sort rasters and/or discard sweeps. See the description of the PSTH in the Analysis menu for details. See `RasterAux()` for the use of auxiliary values.

```
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 flag values: 1 to scale the result as spikes per second, 2 to enable raster displays, 8 for backwards `aCh%` event search. If omitted, 0 is used.
`aCh%`	Auxiliary measurement channel. The measured value sets auxiliary value 0 for sweep sorting. 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 auxiliary value 2, the symbol 1 position. If you omit `aCh%`, there is no auxiliary channel.
`Mn,Mx`	If these two values are present, they set the allowed range of measured values from `aCh%`. If the value for a sweep is outside this range, the sweep is discarded.
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()`. The `Sweeps()` function has no meaning and returns 0.

```
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.
`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 2 to enable raster data, add 4 to enable error bars. 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 of 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 function creates a result view to hold the waveform correlation between two waveform channels, sampled at the same rate. The `Process()` command does the analysis. The function is as follows:

```
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. 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.
`ref%`	The reference waveform channel that is correlated against all the waveform channels.
`bins%`	The number of bins in the correlation. The time taken is proportional to the product of the length of the area processed and the number of bins. This calculation is slow compared to most of the other `SetXXXX` commands. The bin width is set to the sampling interval of the two channels.
`offset`	The length of time to show before the zero time shift in the result, in seconds. If omitted, the value 0.0 is used.
Returns	The function returns a handle for the new view, or a negative error code.

`Sweeps()` returns the number of data points used on the reference channel. Result views saved with this analysis and later restored do not re-link to their time view.

See also:`SetWaveCrlDC()`, `Process()`, `Sweeps()`

SetWaveCrlDC()

This function can only be used after `SetWaveCrl()` has defined the view for a waveform correlation. It sets whether the DC levels of the two signals is removed before the correlation or not. The current view must be a waveform correlation result view.

```
Func SetWaveCrlDC({useDC%})
```

`useDC%`	If present, a zero value removes the DC, a non-zero value (default) includes it.
Returns	The state of using DC before the call. It returns a negative error if this view is a result view but not a waveform correlation.

The correlation can switch between using DC or not at any time once the window has been created. If you change the setting the view is drawn at the next opportunity.

See also:`SetWaveCrl()`, `Process()`

Sin()

This function calculates the sine of an angle in radians, or converts an array of angles into an array of sines.

```
Func Sin(x|x[]);
```

`x`	The angle, expressed in radians, or a real array of angles. The best accuracy of the result is obtained when the angle is in the range -2π to 2π.
Returns	When the argument is an array, the function replaces the array with the sines of all the points and returns either a negative error code or 0 if all was well. When the argument is not an array the function returns the sine of the angle.

See also:`ATan()`, `Cos()`, `Tan()`

Sound()

This has two variants. The first plays a tone of set pitch and duration in Windows NT and a short "beep" in Windows 95. The second plays a `.WAV` file or system sound if your system has multimedia support. The `.WAV` output was added at version 3.04.

```
Func Sound(freq%, dur{, midi%});                                  Tone output
Func Sound(name${, flags%);                             Multimedia sound output
```

`freq%`	If `midi%` is 0 or omitted this holds the sound frequency in Hz. If `midi%` is non-zero this is a MIDI value in the range 1-127. A MIDI value of 60 is middle C, 61 is C# and so on. Add or subtract 12 to change the note by one octave.
`dur`	The sound duration, in seconds. The script stops during output.
`midi%`	If this is present and non-zero, the frequency is interpreted as a MIDI value; otherwise it is a frequency in Hz.
`name$`	Either the name of `.wav` file or the name of a system sound. You can either supply the full path to the file or just a file name and the system will search for the file in the current directory, the Windows directory, the Windows system directory, directories listed in the `PATH` environmental variable and the list of directories mapped in a network. If no file extension is given, `.wav` is assumed. The file must be short enough to fit in available physical memory, so this function is suitable for files of a few seconds duration only.

A blank name halts sound output. If `name$` is any of the following (case is important), a standard system sound plays:

`"S*"`	Asterisk	`"SS"`	System start	`"SE"`	System exit
`"S?"`	Query	`"SW"`	System welcome	`"SD"`	System default
`"SH"`	Hand	`"S!"`	System exclamation		

`flags%` This optional argument controls how the data is played. It is the sum of the following values (given in hexadecimal and decimal):

0x0001	1	Play asynchronously (start output and return). Without this flag, Spike2 does nothing (including sampling) until replay ends.
0x0002	2	Silence when sound not found. Normally `Sound()` plays the system default sound if the nominated sound cannot be found.
0x0008	8	Loop sound until stopped by another Sound command. You must also supply the asynchronous flag if you use loop mode.
0x0010	16	Don't stop a playing sound. Normally, unless the "No wait" flag is set, each command cancels any playing sound.
0x2000	8192	No wait if sound is already playing. `Sound("",0x2010)` can be used to detect if a previous asynchronous sound has finished.

If you don't supply this argument, the flag value is set to 0x2000.

Returns The tone output returns 0 or a negative error code. The multimedia output returns non-zero if the function succeeded and zero if it failed.

Sqrt()

Forms the square root of a real number or an array of real numbers. Negative numbers cause the script to halt with an error when x is not an array. With an array, negative numbers are set to 0 and an error is returned.

```
Func Sqrt(x|x[]);
```

`x`	A real number or a real array to replace with an array of square roots.
Returns	With an array, this returns 0 if all was well, or a negative error code. With an expression, it returns the square root of the expression.

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

Str$()

This converts a number to a string.

`Func Str$(x {,width% {,decs%}});`

`x`	A number to be converted.
`width%`	Optional minimum field width. The number is right justified in this width.
`decs%`	Optional number of decimal places.
Returns	A string holding a representation of the number.

See also:`Asc()`, `Chr$()`, `DelStr$()`, `InStr()`, `LCase$()`, `Left$()`, `Len()`, `Mid$()`, `Print$()`, `Print()`, `Right$()`, `UCase$()`, `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 name and version of the operating system as a string.

`Func System$()`

Returns A string describing the operating system: `"Windows SS build n"` where `SS` is the operating system and `n` is the build number.

See also:`System()`

System()

This function returns the version of the operating system as a number. Use the `App()` command to get the version number of Spike2.

`Func System()`

Returns The operating system revision level times 100. 351=NT 3.51, 400=95 and NT 4, 410=98, 490=Me, 500=NT 2000, 501=XP.

See also:`App()`, `System$()`

Tan()

This calculates the tangent of an angle in radians or converts an array of angles into tangents. Tangents of odd multiples of π/2 are infinite, so cause computational overflow. There are 2π radians in 360°. The value of π is 3.14159265359 (`4.0*ATan(1)`).

`Func Tan(x|x[]);`

`x`	The angle, expressed in radians, or a real array of angles. The best accuracy of the result is obtained when the angle is in the range -2π to 2π.
Returns	For an array, it returns a negative error code (for overflow) or 0. When the argument is not an array the function returns the tangent of the angle.

See also:`ATan()`, `Cos()`, `Ln()`, `Log()`, `Pow()`, `Sin()`, `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 the `TimeDate()` function. 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, a value of 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 procedure returns the time and date in seconds, minutes, hours, days, months, and years. It can also return the day of the week. You can either use separate variables for each field or an integer array of the desired size. This procedure returns numbers. For a formatted string containing either the date or the time, 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(now%[])
```

`s%` If this is the only argument is passed, the number of seconds since midnight is returned in this variable. If the `min%` argument is present, the number of seconds since the beginning of the present minute is returned.

`m%` If this is the last argument, then the number of minutes since midnight is returned in this variable. If `hour%` is present, then the number of full minutes since the beginning of the present hour is returned

`h%` If present, the number of hours since Midnight is returned in this variable.

`d%` If present, the day of the month is returned as an integer in the range 1 to 31.

`mon%` If present, the month number is returned as an integer in the range 1 to 12.

`y%` If present, the year number is returned here. It will be an integer such as 2002.

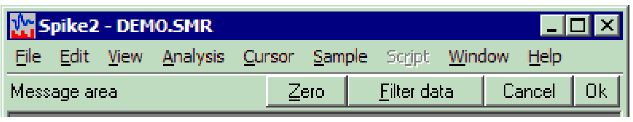
`wDay%` If present, the day of the week will be returned here as 0=Monday to 6=Sunday.

`now%[]` If the only argument is an array, it is filled with time and date data. Elements beyond the seventh are not changed. The array can be less than seven elements long. Element 0 is set to the seconds, 1 to the minutes, 2 to the hours, and so on.

See also:`Date$()`, `MaxTime()`, `Seconds()`, `Time$()`

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.



You can define up to 40 buttons in your toolbar (17 prior to version 4.06), but you will probably be limited by the available space. Buttons are numbered from 1 to 40. There is an invisible button 0, which sets a function that is called when the toolbar is waiting for a button to be pressed.

From version 4.03 onwards, buttons can be linked to the keyboard. However, any keys that are associated with buttons become the exclusive property of the toolbar when it is active and waiting for a button press. If you associate the A key with a button, each time you press that key, the button will be pressed, even if the text caret is in a text window.

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).

See also:`Interact()`, `Toolbar()`, `ToolbarClear()`, `ToolbarEnable()`, `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.

```
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.

The buttons are displayed in order of their item number. Undefined items leave a gap between the buttons. This effect can be used to group related buttons together.

See also:`Interact()`, `ToolbarClear()`, `ToolbarEnable()`, `ToolbarSet()`, `ToolbarText()`, `ToolbarVisible()`

ToolbarClear()

This function is used to remove all, or some of the buttons from the toolbar. If you delete all the buttons the `Toolbar()` function will insert a button labelled OK so you can get out of the `Toolbar()` function. Use `ToolbarText("")` to clear the toolbar message.

```
Proc ToolbarClear({item%});
```

`item%`	If present, this is the 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()`, `ToolbarEnable()`, `ToolbarSet()`, `ToolbarText()`, `ToolbarVisible()`

ToolbarEnable()

This function enables and disables toolbar buttons, and reports on the state of a button. Enabling an undefined button has no effect. If you disable all the buttons and then use the `Toolbar()` function or if you disable all the buttons in a function linked to the toolbar, and there is no idle function set, a single OK button is displayed.

```
Func ToolbarEnable(item% {,state%});
```

`item%` The number of the button or -1 for all buttons. You must enable and disable button 0 with `ToolbarSet()` and `ToolbarClear()`.

`state%` If present this sets the button state. 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()`

ToolbarSet()

This function adds a button to the toolbar and optionally associates a function with it. When a button is added, it is added in the enabled state.

```
Proc ToolbarSet(item%, label$ {,func ff%()});
```

`item%` The button number to add or replace. Buttons are numbered from 1 to 40 (1 to 17 prior to version 4.06). You can use an item number of 0 to set or clear a function that is called repeatedly while the toolbar waits for a button press.

You can link a button to the Esc key, by negating `item%`. For example `ToolbarSet(-2,"Quit");` sets button 2 as the escape key.

`label$` The label for the button. Labels compete for space with each other, so it is a good idea to keep the labels fairly short. The label is ignored for button 0.

You can link a key to a button by placing `&` before a character in the label, or by adding a vertical bar and a key code in hexadecimal (e.g. 0x30), octal (e.g. 060) or decimal (e.g. 48) to the end of the label. Characters set by `&` are case insensitive. For example `"a&Maze"` generates the label aMaze and responds to `m` or `M`; the 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 separator
0x6d nk -	0x62 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. This function should have no arguments and the name with no brackets is given, for example `ToolbarSet(1,"Go",DoIt%);` where `Func DoIt%()` is defined somewhere in the script. When the `Toolbar()` function is used and the user clicks on the button, the linked function is run (or if the `item%` 0 function is set, the function runs while no button is pressed). The function return value controls the action of `Toolbar()` after a button is pressed.

If it returns 0, the `Toolbar()` function returns to the caller, passing back the button number. If it returns a negative number, the `Toolbar()` call returns the negative number. If it returns a number greater than 0, the `Toolbar()` function does not return, but waits for the next button. An item 0 function must return a value greater than 0, otherwise `Toolbar()` will return immediately.

If this argument is omitted, there is no function linked to the button. When the user clicks on the button, the `Toolbar()` function returns the button number.

See also: `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()`

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

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 cased version of the original string.

See also: `Asc()`, `Chr$()`, `DelStr$()`, `InStr()`, `LCase$()`, `Left$()`, `Len()`, `Mid$()`, `Print$()`, `ReadStr()`, `Right$()`, `Str$()`, `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%});
```

`text$` The string to convert to a number. The expected format is:
`{white space}{-}{digits}{.digits}{e|E{+|-}digits}`

`nCh%` If present, it is set to the number of characters used to construct the number.

Returns It returns the extracted number, or zero if no number was present.

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

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 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)`. You can use `View(-1)` in a result or XY view to refer to the parent time view.

Returns the view 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 the function that follows the dot. For example, `View(vh%).Draw(1,2)` draws view number vh%. It is an error if the selected view does not exist, and the script stops. If the function closes the original view, the script returns to the next available view.

```
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 array that forms a result view channel. In version 4, the syntax is extended to include a channel number. In previous versions, result views had a single channel.

```
View(vh%{,ch%}).[]
```

`vh%` A view handle of an existing result view or 0 for the current result view.

`ch%` An optional channel number in the result view. If omitted, channel 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()`

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 title that matches the string, or 0 if no view matches the title.

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 and the sequencer control panel.

```
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 `n`th 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()`, `View()`, `ViewList()`

ViewList()

This function fills an integer array with a list of view handles. It never returns the view handle of the running script; use the `App()` command to get this.

```
Func ViewList({list%[] {,types%}});
```

`list%` An integer array that is returned holding view handles. The first element of the array (element 0) is filled with the number of handles returned. If the array is too small to hold the full number, the number that will fit are returned.

`types%` The types of view to include. This is a code that can be used to filter the view handles. The filter is formed by adding the types from the list below. If this is omitted or if no types are specified for inclusion, all view handles are returned.

1	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()`

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. It has no effect on other view types.

```
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.

`Func ViewTrigger(chan%, pre, hold, xZero% {,cur0%});` Set and enable
`Func ViewTrigger({mode%});` Enable/disable and get information

`chan%`	The event, marker, WaveMark, TextMark or RealMark trigger channel.
`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 optional argument is new in version 4 and controls the cursor 0 action each time the view is triggered. 0=no action, 1=move cursor 0 without causing the active cursors to iterate, 2=move cursor 0 and update any active cursor positions. If this argument is omitted, the value 0 is used.
`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	Move to the next trigger and return the trigger time or -1 if no trigger is found or if the trigger is not enabled.
2	Move to the previous 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.
0	Disable the trigger.
-1	Return the trigger channel.
-2	Return the pre trigger time.
-3	Return the hold time.
-4	Return the `xZero%` state.
-5	Return the `cur0%` state.

Returns When the command is used with a negative argument, the command returns the requested information, of -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()`, `ViewStandard()`, `XYDrawMode()`

ViewUseColour()

This function can be used to force the display to use black and white only, or to use the colours set in the Colour dialog or by the `Colour()` command.

```
Func ViewUseColour( {use%} );
```

`use%`	If present, a value of 0 forces 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()`

Window()

This sets the position and size of the current view with respect to the application window. Positions are from the top left-hand corner as percentages of the application window size. This can also be used to position, dock and float dockable toolbars.

```
Proc Window(xLow, yLow{, xHigh{, yHigh}})
```

`xLow` Position of the left hand edge of the window. When docking a dockable toolbar, the `xLow` and `xHigh` values correspond to the position of the top left corner of the window when dropped with the mouse. You will need to experiment with both values to get multiple bars to dock together tidily.

`yLow` Position of the top edge of the window.

`xHigh` If present, the right hand edge. If omitted the previous width is maintained. If the window is made too small, the minimum allowed size is used. If the current view is a dockable control and `yHigh` is 0, values less than 1 or greater than 4 float the window at `(xLow, yLow)`, otherwise `xHigh` sets the docking state:

1	Docked to the left window edge	3	Docked to the right edge
2	Docked to the top window edge	4	Docked to the bottom edge

`yHigh` If present, the bottom edge position. If omitted the previous height is maintained. If the window is made too small, the minimum allowed size is used.

If the Window is dockable and `yHigh` is 0, this command sets the docked state of the window (see `xHigh`) otherwise the window is floated with the nearest allowed width that is no more than `xHigh-xLow`. If `xHigh-xLow` is 0 or negative `YHigh` sets the height of the dockable window.

This can be used to position the application window in Windows (not Macintosh):

```
view(App()).Window(0,0,100,100);     'set maximum application size
```

See also:`App()`, `WindowDuplicate()`, `WindowGetPos()`, `WindowSize()`, `WindowTitle$()`, `WindowVisible()`

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 no free duplicates. There is a limit of 19 duplicates per window (9 before version 3.11).

See also:`Window()`, `WindowGetPos()`, `WindowSize()`, `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)
```

`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.

See also:`Window()`, `WindowDuplicate()`, `WindowSize()`, `WindowTitle$()`, `WindowVisible()`

WindowSize()

This resizes the current view without changing the top left-hand corner position. Setting a window dimension less than zero leaves the dimension unchanged. Setting a dimension smaller than the minimum allowed sets the minimum value. Setting a size greater than the maximum allowed sets the maximum size. There are no errors from this function.

```
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 use this to resize the application window for systems that support this (Windows). If the system does not support it, no change is made.

You can also use this to resize a dockable control bar when it is floating (use `App()` to get the window handle). In this case, 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()`, `WindowDuplicate()`, `WindowGetPos()`, `WindowTitle$()`, `WindowVisible()`

WindowTitle$()

This function gets and sets the title of the current window. There may be windows that are resistant to having their title changed. For these, the routine has no effect. Most windows can return a title. If you change a title, dependent window titles change, for example, cursor windows belonging to 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()`

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.

Returns The window state prior to this call.

See also: `FrontView()`, `Window()`, `WindowDuplicate()`, `WindowGetPos()`, `WindowSize()`, `WindowTitle$()`

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%` Optional, set the axis state. If omitted, no change is made. Possible values are:

0 Hide the axis
1 Show the 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()`

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 mode (seconds, hours minutes and seconds, time of day) for a time view, and the major and minor tick spacing for all views that have an x axis. If you set values that would cause illegible axes, they are stored but not used unless the axis range or scaling changes to make the values useful.

```
Func XAxisStyle({style%{, nTick%{, major}}});
```

`style%`	In a time view, set 1 for an axis in seconds, 2 for hours minutes and seconds and 3 for time of day. Omit `style%` or use 0 to leave the style unchanged. A value of –1 returns the number of minor divisions set or 0 for automatic. A value of –2 returns the major tick spacing or 0 for automatic spacing.
`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.

```
Func XHigh()
```

Returns In a time view, the result is in seconds. In a result view, the result is in bins and can be fractional. In an XY view the result is in x axis units.

It is a fatal error to use this in an inappropriate view. The following code will page through a time or result view from the current position to the end of the file.

```
while (xHigh() < MaxTime()) do    'while not reached the end
   Draw(XHigh())                  'draw from next page
   wend;                          'end of while loop
```

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.

```
Func XLow()
```

Returns In a time view, the result is in seconds. In a result view, the result is in bins and can be fractional. It is a fatal error to use this in an inappropriate view.

For example, this code pages a time or result view from the current position to the start of the file. In an XY view it moves the view to the left if the current position is past 0.

```
while (XLow() > 0) do             'while not at 0
   Draw(XLow()-(XHigh()-XLow()))  'draw previous page
   wend;                          'end of while loop
```

See also:`Draw()`, `XRange()`, `BinToX()`, `XToBin()`, `XHigh()`

XRange()

This sets the start and end of the x axis in a time result or XY view in x axis units (not bins for a result view). Unlike `Draw()`, it does not update the view immediately; updates must wait for the next `Draw()`, `DrawAll()` or some interactive activity.

```
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 general, this will not be an integral number of bins, however, when used to access a bin, it will be truncated to an integer, and will refer to the bin that contains the x value.

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 one or more data points to a channel in an XY view. 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.

```
Func XYAddData(chan%, x|x[]|x%[], y|y[]|y%[]);
```

`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). 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).
Returns	The number of data points which have been added successfully.

See also:`XYColour()`, `XYDelete()`, `XYDrawMode()`, `XYJoin()`, `XYKey()`, `XYRange()`, `XYSetChan()`, `XYSize()`, `XYSort()`

XYColour()

This gets or sets the colour of a channel in the current XY view. The default channel colour is black.

```
Func XYColour(chan% {,col%});
```

`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; their indexes are numbered from 0 to 39. If omitted, there is no colour change.
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()`, `XYJoin()`, `XYGetData()`, `XYInCircle()`, `XYInRect()`, `XYRange()`, `XYSetChan()`, `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 relevant channel draw mode or axis expanding mode at time of call if succeeds or a negative error code if fails.

See also:`XYAddData()`, `XYColour()`, `XYCount()`, `XYDelete()`, `XYGetData()`, `XYInCircle()`, `XYInRect()`, `XYJoin()`, `XYKey()`, `XYRange()`, `XYSetChan()`, `XYSize()`, `XYSort()`

XYGetData()

This gets data points between two indices from a channel in the current XY view. It is an error to use this command with any other view type.

```
Func XYGetData(chan%, &x|x[], &y|y[] {,first% {,last%}});
```

`chan%` A channel number in the current XY view. 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, the first data point is index 0.

`last%` The zero-based index of the last data point returned. `last%` is only meaningful when `x` and `y` are array names. If omitted or if `last%` is greater than or equal to the number of data points, the final data point is the last one in the channel. If `last%` is less than `first%`, no data points are returned.

Returns The number of data points copied. This can be less than the number of data points between the requested indices. For example, if the size of x or y array is not big enough to hold all the data points from `first%` to `last%`, the number of data points returned is equal to the size of array. If x and y are simple variables, 1 is returned if the data point with index number `first%` exists.

The index number of a data point depends on the current sorting method (see `XYSort()`).

See also: `XYAddData(), XYColour(), XYCount(), XYDelete(), XYDrawMode(), XYInCircle(), XYInRect(), XYJoin(), XYRange(), XYKey(), XYSetChan(), XYSize(), XYSort()`

XYInCircle()

This gets the number of data points inside a circle defined by `xc`, `yc`, and `r` in the current XY view. A general point `(x,y)` is considered to be inside the circle if:

$(x-xc)^2 + (y-yc)^2 <= r^2$

Points lying on the circumference are considered inside, but due to floating point rounding effects they may be indeterminate.

```
Func XYInCircle (chan%, xc, yc, r);
```

`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.
Returns	the number of data points inside the circle or a negative error code.

See also: `XYAddData(), XYColour(), XYCount(), XYDrawMode(), XYGetData(), XYInCircle(), XYInRect(), XYJoin(), XYRange(), XYKey(), XYSetChan(), XYSize(), XYSort()`

XYInRect()

This function returns the number of data points in a channel of the current XY view that lie inside a rectangle. Data points on the boundaries of the rectangle are considered to be inside, but due to floating point rounding they may be indeterminate.

```
Func XYInRect (chan%, xl, yl, xh, yh);
```

`chan%`	A channel number in the current XY view.
`xl,xh`	The `x` co-ordinates of the left and right hand edges of the rectangle. `xh` must be greater than or equal to `xl`.
`yl,yh`	The `y` co-ordinates of the bottom and top edges of the rectangle. `yh` must be greater than or equal to `yl`.
Returns	The number of data points inside the rectangle or a negative error code.

See also: `XYInCircle(), XYDelete(), XYDrawMode(), XYGetData(), XYJoin(), XYRange(), XYSetChan(), XYSize(), XYSort()`

XYJoin()

This function gets or sets the data point joining method of a channel in the current XY view. Data points can be separated, joined by lines, or joined by lines with the last point connected to the first point (making a closed loop).

```
Func XYJoin(chan% {,join%});
```

`chan%` A channel number in the current XY view. The first channel is number 1. –1 is also allowed, meaning all channels.

`join%` If present, this is the new joining method of the channel. If this is omitted, no change is made. The data point joining methods are:

0 Not joined by lines (this is the default joining method)
1 Joined by lines. The line styles are set by `XYDrawMode()`.
2 Joined by lines and the last data point is connected to the first data point to form a closed loop.

Returns the joining method at time of call if succeeds or a negative error code if fails.

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()`

XYRange()

This function gets the range of data values of a channel or channels in the current XY view. This is equivalent to the smallest rectangle that encloses the points.

```
Func XYRange(chan%, &xLow, &yLow, &xHigh, &yHigh);
```

`chan%` A channel number in the current XY view or -1 for all channels or -2 for all visible channels. The first channel is number 1.

`xLow` A variable returned with the smallest x value found in the channel(s).

`yLow` A variable returned with the smallest y value found in the channel(s).

`xHigh` A variable returned with the biggest x value found in the channel(s).

`yHigh` A variable returned with the biggest y value found in the channel(s).

Returns 0 if there are no data points, or the channel does not exist, 1 if values found.

See also:`XYAddData()`, `XYCount()`, `XYDrawMode()`, `XYGetData()`, `XYInCircle()`, `XYInRect()`, `XYSetChan()`, `XYSize()`

XYSetChan()

This function creates a new channel or modifies an existing channel in the current XY view. It is an error to use this function if the current view is not an XY view. This function can be used as a short-cut method for modifying all properties of an existing channel without calling the `XYSize()`, `XYSort()`, `XYJoin()` and `XYColour()` commands individually.

```
Func XYSetChan(chan% {,size% {,sort% {,join% {,col%}}}});
```

`chan%` A channel number in the current XY view. If `chan%` is 0, a new channel is created. Each XY view can have maximum of 256 channels, numbered 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 is the available memory and the time taken to draw the view.

A value of zero (the default) sets no limit on the number of points and the size of the channel expands as required to hold data added to it.

If a negative size is given, for example `-n`, this limits the number of points in the channel to `n`. If more than `n` points are added, the oldest points added are deleted to make space for the 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 the last data point is connected to the first data point to form a closed loop.

`col%` If present, this sets the index of the colour in the colour palette to use for this channel. There are 40 colours in the palette, their index numbered 0 to 39. If omitted, the colour of an existing channel is not changed. The default colour for a new channel is the colour that a user has chosen for an ADC channel in a time window.

Returns The number of channels (including any created channel) or a negative error code. When you create a channel, the value returned is the new channel number.

See also: `XYAddData()`, `XYColour()`, `XYDelete()`, `XYDrawMode()`, `XYJoin()`, `XYKey()`, `XYRange()`, `XYSize()`, `XYSort()`

XYSize()

This function gets and sets the limits on the number of data points of a channel in the current XY view. Channels can be set to have a fixed size, or to expand as more data is added. The only limit on the number of data points is the available memory and the time taken to draw data.

```
Func XYSize(chan% {,size});
```

`chan%` A channel number in the current XY view. 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 added 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()`, `XYColour()`, `XYCount()`, `XYDelete()`, `XYDrawMode()`, `XYGetData()`, `XYInCircle()`, `XYInRect()`, `XYJoin()`, `XYKey()`, `XYRange()`, `XYSetChan()`, `XYSort()`

XYSort()

In the current XY view, gets or sets the sorting method of the channel.

```
Func XYSort(chan% {,sort%});
```

`chan%` the channel number 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()`, `XYColour()`, `XYCount()`, `XYDelete()`, `XYDrawMode()`, `XYGetData()`, `XYInCircle()`, `XYInRect()`, `XYJoin()`, `XYKey()`, `XYRange()`, `XYSetChan()`, `XYSize()`

YAxis()

This function is used to turn the y axes on and off, in the current view and to find the state of the y axes in a view.

`Func YAxis({on%});`

`on%`	Optional argument that sets the state of the axes. If omitted, no change is made. Possible values are: 0 Hide all y axes in the view. 1 Show all y axes in the view.
Returns	The state of the y axes at the time of the call (0 or 1) or a negative error code.

See also: `ChanOffset()`, `ChanScale()`, `Grid()`, `Optimise()`, `XAxis()`, `XScroller()`, `YAxisMode()`, `YHigh()`, `YLow()`, `YRange()`

YAxisMode()

This function controls what is drawn in a y axis and where the y axis is placed with respect to the data.

`Func YAxisMode({mode%});`

`mode%` Optional argument that controls how the axis is displayed. If omitted, no change is made. Possible values are the sum of the following values:

1	Hide all the title information.
2	Hide all the unit information.
4	Hide y axis small ticks. They are also hidden when big ticks are hidden.
8	Hide y axis numbers. They are also hidden when big ticks are hidden.
16	Hide the big ticks and the vertical line that joins them.
32	Scale bar axis. If selected add 4 to remove the end caps.
4096	Place the y axis on the right of the data

Returns The state of the y axis mode at the time of the call or a negative error code.

See also: `ChanNumbers()`, `YAxis()`, `YAxisStyle()`, `YHigh()`, `YLow()`, `YRange()`

YAxisStyle()

This function controls the y axis major and minor tick spacing. If you set values that would cause illegible or unintelligible axes, they are stored but not used unless the axis range or scaling changes to make the values useful.

`Func YAxisStyle(cSpc, opt%{, major});`

`cSpc`	A channel specifier or -1 for all, -2 for visible and -3 for selected channels. When multiple channels are specified, returned values are for the first channel.
`opt%`	Values greater than 0 set the number of subdivisions between major ticks. 0 sets automatic small tick calculation. Use –1 for no change. Values less than -1 return information, but do not change the axis style
`major`	If present and `opt%` is greater than -2, values greater than 0 sets the major tick spacing. Values less than or equal to 0 select automatic major tick spacing.
Returns	If `opt%` is –2 this returns the current number of forced subdivisions or 0 if they are not forced. If `opt%` is –3 this returns the current major tick spacing if forced or 0 if not forced. Otherwise the return value is 0 or a negative error code.

See also: `YAxis()`, `YAxisMode()`, `YHigh()`, `YLow()`, `YRange()`, `XAxisStyle()`

Yield()

This suspends the script and allows the system to idle and do hosekeeping. During the idle time, invalid screen areas update, you can interact with the program. If your script runs for long periods without using `Interact()` or `Toolbar()`, adding `Yield()` to loops may make it feel more responsive and stop the operating system marking Spike2 as "not responding". This function was added at version 4.07.

```
Func Yield({wait{, allow%}});
```

`wait`	An optional time period, in seconds, to wait. If omitted or set to 0, the program will give the system one idle cycle and continue to run. If set negative, there is no idle cycle but the `allow%` argument is applied.
`allow%`	This defines what the user can do during the wait. See `Interact()` for allowed values. The 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()`, `TimeDate()`, `Toolbar()`

YHigh()

This function returns the upper y axis limit for the y axis in a time, result or XY view.

```
Func YHigh(chan%);
```

`chan%`	The channel number (use 0 for a result or XY view).
Returns	The value at the appropriate end of the axis.

See also:`YLow()`, `YRange()`, `ChanScale()`, `ChanOffset()`, `Optimise()`

YLow()

This function returns the lower y axis limit for the y axis in a time or result view.

```
Func YLow(chan%);
```

`chan%`	The channel number. Use 0 for a result or XY view.
Returns	The value at the appropriate end of the axis.

See also:`YHigh()`, `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 is invalidated, but waits for the next `Draw()`. If you omit the `low` and `high` arguments, Spike2 will display the “full” range of the channel, equivalent to the Show All button in the Y Axis dialog.

```
Proc YRange(chan%, {low, high})
```

`chan%`	The channel number in a time view, 0 in a result view. You can also use -1 for all channels, -2 for visible channels and -3 for selected channels. The channel number is ignored in an XY view as there is only one axis for all channels.
`low`	The value for the bottom of the y axis. If omitted, and the channel type has a known range, Spike2 will set the `low` and `high` values to suitable limits. For example, for a waveform channel displayed as a waveform the limits are those implied 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:`YHigh()`, `YLow()`, `ChanScale()`, `ChanOffset()`, `Optimise()`

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 `PROC`s translate without any problem as do expressions using these items.

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.

What will not translate

We cannot translate concepts that have no equivalent in the windowed environment. For example, overdrawing of data has no equivalent (yet) so does not translate. 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.

- 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 may be able to emulate overdrawing with an XY view. The overdraw WaveMark mode will overdraw Spike shapes for you.
`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.
- 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 version 4 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
```

The 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:

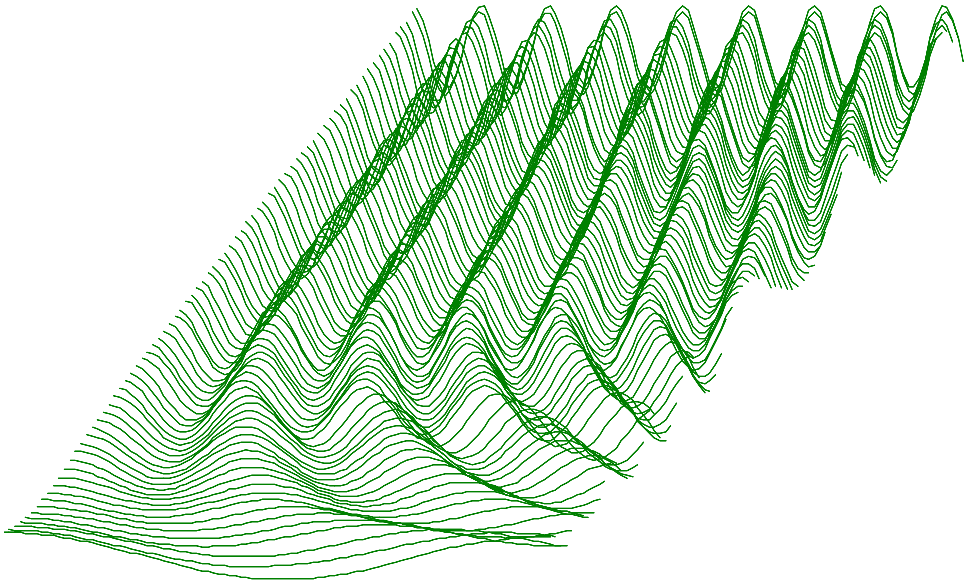
```
XYAddData(1, 0, 0);         'add a point to channel 1 at (0, 0)
XYAddData(1, 1, 0);         'add a point at (1,0)
XYAddData(1, 0.5, 1);       'add a point at (0.5, 1)
```

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;
```

Index